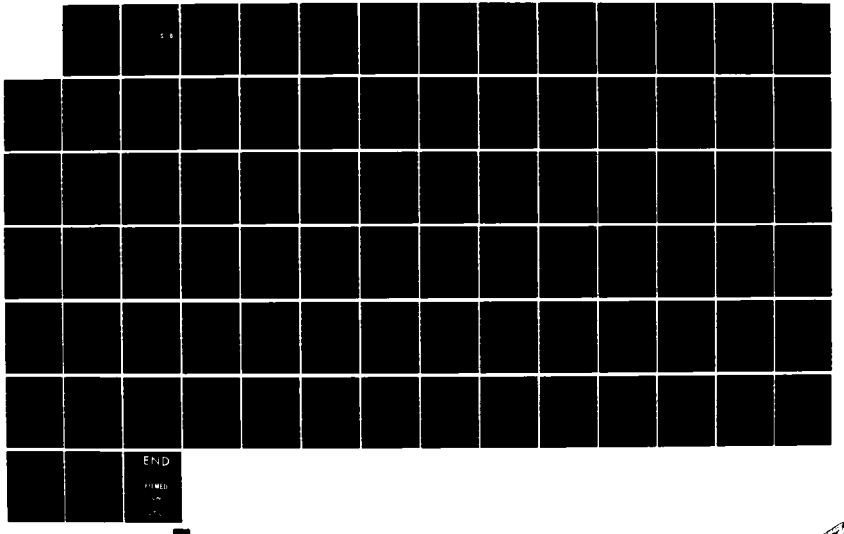
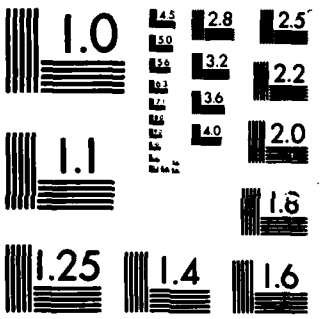
AD-A164 265 FORMALIZATION OF THE PROGRAM REFERENCE LANGUAGE(U) 1/1
ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW
CA W M BRICKEN ET AL. 08 OCT 85 AI/DS-TR-1066-01
UNCLASSIFIED AFOSR-TR-85-1219 F49620-84-C-0075 F/G 9/2 NL





AI.DS
ADVANCED INFORMATION
& DECISION SYSTEMS

201 San Antonio Circle, Suite 286
Mountain View, CA 94040
(415) 941-3912

TR-1066-01

FORMALIZATION OF THE PROGRAM REFERENCE LANGUAGE

Prepared by:

William M. Bricken
Susan G. Rosenbaum
Michael A. Brzustowicz
Jeffrey S. Dean
Brian P. McCune

8 October 1985

Final Report for July 1984 to September 1985

Contract Number: F49620-84-C-0075

Approved for Public Release: Distribution Unlimited

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling Air Force Base, D.C. 20332-6448

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

ADA 164264

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): TR-1066-01	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 85-1219
6a. NAME OF PERFORMING ORGANIZATION: Advanced Information & Decision Systems; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: AFOSR/NM
6c. ADDRESS (City, State and ZIP Code): 201 San Antonio Circle, Suite 286, Mountain View, CA 94040-1270	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical and Information Sciences, Building 410, Bolling AFB, D.C. 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: same as #7; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-84-C-0075
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	3304	A2	

11. TITLE (Include Security Classification): Formalization of the Program Reference Language

12. PERSONAL AUTHOR(S): Bricken, William M.; Rosenbaum, Susan G.; Brzustowicz, Michael A.; Dean, Jeffrey S.; McCune, Brian P.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 84/07/15 TO 85/09/14	1985 October	85

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
05	08	
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Program Reference Language (PRL), Extended Program Model (EPM), Intelligent Program Editor (IPE), Artificial Intelligence (AI), program editing, Ada editor, Ada syntax, semantic model, pictorial logic, query language, LOSP.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The goal of the Program Reference Language (PRL) Project is to construct a representation of Ada programs that facilitates retrieval of code based on both syntactic (literal) and semantic (functional) queries. The fourth year of the project focused on the formalization of the Extended Program Model (EPM), which consists of textual, syntactic, and semantic representations. The PRL query language specifies search over these three interrelated databases. Textual items are retrieved by string-matching capabilities of standard editors; syntactic queries are directed to the syntax parse tree; queries referencing program functionality are mapped onto the LOSP semantic representation.

Currently, we have implemented a generic Parser-Developer, that builds a syntactic parser for a language given the grammar, and a Mock-Interface Engine for rapid configuration of interface designs. Design studies of the query language, the efficiency of retrieval from the EPM, and the user interface have been conducted.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Captain John Thomas	(202) 767-5025	NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

Table of Contents

List of Figures

1. Introduction

This report documents the fourth year of work on the Program Reference Language (PRL) Project at Advanced Information & Decision Systems.

The objective of this project is to provide the user with a tool that permits intelligent access to existing program code. The PRL is designed to reference programs based on the textual representation of the code (text), on the rules of combination for the textual atoms (syntax), and on the behavior of these combinations when interpreted (procedural semantics).

To accomplish this degree of access to the representation of the code, the PRL relies upon multiple models of the program. This collection of models, called the Extended Program Model (EPM), consists of a literal (textual) representation, a grammarical (syntactic) parse tree of the code, and a computational (semantic) model. The semantic model stores variable and constant tokens in a representational space that is independent of the representation of control, while maintaining functional invariance. We believe that searching and querying the EPM can be achieved efficiently. The design of the PRL includes a Pictorial Language (PL) and a sophisticated user-interface.

The long-term goals of the PRL Project are:

1. to construct a representation of program text that incorporates both syntactic and semantic information,

2. to construct a query language that is easy to use, allowing the user to specify program fragments with both syntactic and semantic descriptions,

3. to devise search mechanisms so that queries are satisfied efficiently,

4. to accommodate program editing and manipulation, and

5. to present these facilities to the user in a satisfying and informative interface package.

The goals that have been achieved during the research period from July 15, 1984 to July 14, 1985 are:

1. the formal specification of most of the Extended Program Model, including the implementation of the syntactic model and the design of the semantic model,

2. the design and formal specification of a sequence of automated construction algorithms that convert a substantial subset of Ada code into the EPM,

3. the study of efficient search mechanisms for this system,

4. the design and specification of an intermediate query language that permits access to the EPM,

5. the design of a friendly, yet formal Pictorial Language (PL) that is compatible with the intermediate query language, and

6. the design of a user-interface that incorporates principles of convenience, understandability, responsiveness, and formal consistency.

From the above list, it is apparent that this research year has been pivotal to the PRL Project. Specifically, we have accumulated enough experience with the problem to be able to transfer from a study phase to an implementation phase. Particularly important to this year's effort is the commitment to formalization and mechanization of two major components of the PRL: the Extended Program Model and the formal Picture Language.

Significant software tools developed during the last year include:

1. The Parser-Developer: constructs a syntactic parser for a given language when a grammer for that language is provided.

2. The Interface Prototyper: an environment for rapid prototyping of the visual display of the PRL user-interface.

During future phases of this research, we envisage a capability that not only allows the user to specify search for code at a mixed syntactic and semantic level, but also is able to suggest stylistic revision of code that will decrease debugging time, increase clarity of code comprehension, and specifically detect portions of code receptive to automated verification.

A summary of the previous three year's work is available in the technical reports

for the previous years of this project. *Design of an Intelligent Program Editor* (TR-3023-1, Sept. 1982) contains initial design considerations and description of a prototype system. *Design of a Pictorial Program Reference Language* (TM-1014-4, Aug. 1984) includes the description of an informal version of the picture language.

This report addresses design and specification of three major components of the PRL:

1. the Extended Program Model, the database of program representations accessed by the PRL.

2. the Picture Language, a pictorial representation of queries that is both easy to use and formally equivalent to Propositional Logic.

3. the User-PRL Interface.

The following technical chapters describe the specifications of the Extended Program Model, the intermediate query language, the Picture Language, and our design considerations for searching the EPM and for presenting the system to the user.

2. The Extended Program Model

The Extended Program Model (EPM) is a set of representations of program source code. It is central to the PRL since it is the stored representation that is the query database. Were the source code stored solely as text, only textual queries would be possible. This is this case for simple editors. The EPM, however, is intended to permit queries not only of textual structure, but also of syntactic and semantic structure in the code. For this reason, the EPM is an extended representation; it models the code at several abstract levels. Naturally, different abstractions require different representations. The EPM currently includes three abstract representations: the textual copy, the syntactic parse tree, and the semantic model. We are able to automate the construction of these representations from the source code. A fourth model, the documentation text model, includes text that is not an integral part of the code itself.

Our previous research has identified yet another abstract level of representation, that of programming cliches, or typical programming patterns. These are larger chunks of the semantic representation that identify common patterns in programs, such as collecting tokens, linked list manipulations, search strategies, etc. We have not formalized a definition of these programming cliches; instead our effort during the past year has been to concentrate on rigorous formalization of lower level abstractions.

The documentation level contains comments and notations excluded from the functional representation of the code. Documentation can serve multiple purposes:

- to describe the operation of the code;
- to describe the organizational structure of the code;
- to describe the connectivity of the code, its interface with other modules;
- to describe the intended uses of the code, its pragmatics.

The long term design of the EPM includes incorporation of the model of the code conceptualized by the author, in the form of structured documentation. This model will permit semantic correction and guidance by the editor while the user is constructing code from his specification/model. The design of the documentation model is not discussed in this report.

We present our technical approach to the definition of the EPM in the next Section. Following that is a discussion of the formal models embodied in the EPM. As yet we have automated only portions of this process; we expect to have full automation within the next research year.

2.1 The Modelling Goals

Our goals for developing the EPM are:

1. to design a fully automated conversion process that constructs the different models of the EPM from the source code,

2. to maintain a formal rigor in the specification of the component models, and

3. to assure that search, retrieval, and modification of the EPM database is both possible and efficient.

The technical characteristics that we wish to incorporate into the EPM are:

1. All control and reference must be explicit.

2. The complexity of the representation must be *minimal*, while the functionality of the code must be invariant, and

3. Following the lead of logic programming languages, control and description of the code must be expressed independently of one another.

Our design and formal specifications for the EPM achieve these technical objectives. However, we found it necessary to partition the problem and to approach smaller portions rather than addressing the entirely general case. Therefore we have adopted two research constraints:

1. We chose to use LISP as the basic research language. This decision is not unprecedented; LISP is the primary research language of the AI community. To illustrate the connectability of the EPM to other languages, we are developing an Ada to LISP conversion program. We chose Ada as our target language due to its prevalence in the military community.

2. We chose to limit our initial implementation to basic programming constructs, specifically: assignment, conditionals, and repetitive application (iteration and recursion).

2.2 The EPM Models

The three MODELS incorporated in the design of the EPM are the Text, the Syntactic and the Semantic Models.

1. The TEXT is a literal model, its representation is a sequence of tokens and token separators.

2. The SYNTACTIC model is a tree representation of the textual source incorporating atomic structures (both names and reserved tokens) as nodes, and syntactic relations as arcs.

   This syntactic parse tree provides structural information about the representation rules of the source language. In our examples, this language is Ada. By creating a syntactic parser for an alternative language, we can map program representations from different languages onto a canonical form. That is, the EPM is generic across languages. To modify it for other languages, all that is needed is the formal grammar of the language, (for the syntactic parser), and a grammar to canonical form mapping, to construct the functionally equivalent expression in the canonical research language. Naturally only subsets of each target language can be accommodated by the EPM. The EPM is not complete over all languages.

3. The SEMANTIC model represents the control structure of the textual source, the data structure of the source, and the functional relations between data and control.

   The model of procedural semantics provides both the power and the uniqueness of the PRL paradigm, since it allows the user to express queries in terms of the functionality of the code, rather than solely in terms of the textual structure.

The textual representation forms the basis from which the syntactic and the semantic models are built. The following Chapter describes the construction of these models. We have completed the implementation of the Parser-Developer, which constructs the syntactic model from the source code and its knowledge of the source language. The design of the tools to build the semantic model is a major focus of the coming year's work. The semantic model itself is outlined in Section 3.4 of the next Chapter.

3. The Syntactic and Semantic Models

The design and implementation of the syntax view of programs is described in this Chapter. We address these aspects of the SYNTACTIC model:

1. the representations chosen for the syntactic model and its closely connected textual model.

2. the implementation of the Parser-Developer, which constructs a parser for a target language. The machine written parsing code, in turn, builds the syntactic model from the text.

3. linking the textual and the syntactic models.

Section 3.1 describes the representation chosen for the syntactic model. This model must be generated automatically from the original program text. In addition, as changes are made to either the program text or to the syntactic model, the other one must be updated correspondingly. The eventual capability of maintaining database consistency when a model is modified is made easier by choosing to represent the textual model as a series of segments, each of which is meaningful in the syntactic model. The textual model is segmented into token strings which correspond directly to the leaf nodes in the syntax tree. Characters not included in any token do not convey meaning; changing these characters to others does not affect the syntactic model. Similarly, every syntax node can be mapped into a set of text segments that represent it. This choice of closely coupled representations facilitates keeping both representations in synchronization with each other, allowing for the future possibility of incremental parsing. It also does not preclude any useful editing features on either model.

Section 3.2 describes how we have implemented the representation of the syntax of a program. It focuses heavily on the parser-developer, which is the key piece of human written software in the system. The parser-developer, in conjunction with other software, writes a surface parser for a language given an accepting grammar for that language. It makes heavy use of the YACC compiler-compiler, which is supplied with the UNIX operating system. The surface parser, when invoked from the control regime of the PRL, creates a data stream from which both a syntax tree and a corresponding segmented text list can be built and linked.

Section 3.3 discusses how the textual and syntactic models are actually linked together, what capabilities are supported by the joint structure, and possible future work.

Finally Section 3.4 introduces an outline of the design of the semantic model for the EPM.

3.1 The Syntactic Model

The syntactic model is closely linked to the textual model, both enhancing the users' ability to comprehend the system and allowing the system to properly maintain consistency between the two representations.

The key issues in the choice of representations for both models are the need to have them closely coupled and to have them usable. Because the user can potentially modify either one or both models, in order to maintain consistency it is desirable that the representations are bidirectionally linked. In addition, if the syntactic model is not going to redundantly include all the program text, then it is necessary from the standpoint of usability that one can make the mappings visible on demand. This also suggests having explicit bidirectional links.

It is easy to see how syntax can point back to text. As the text is being tokenized as part of the parsing process, it is a relatively minor task to instrument the character read routines to provide the offset for the beginning and end of each token in the input stream. But, unless the text is segmented, it makes no sense to talk about a link from the text to the syntax representation. If the program text is a single string (or stream), then that one sting is linked to *all* syntax nodes, which is not useful. The final representation must have links that have *meaning*.

Most editors represent text in one of a few, well known ways. Representing a program by a single string (a limiting case presented above) is an inconvenient strategy, and is not one of the widely accepted representations. Of the commonly accepted approaches, the one that comes closest to the single string representation is the text plus gap editors. Another common approach is to segment the text into lines or characters.

Maintaining a program as a linked list of characters is generally considered to be too resource intensive for practical use. And for our purposes, lines of text do not mesh naturally with the notion of syntactic structure. A given syntax node can refer to a portion of a line, or to portions of several lines, but will, in practice, rarely point to one or more complete lines. The natural solution is to segment the text in a way meaningful to the syntactic model, namely into tokens and token separators.

This is the text representation that we chose; it fits naturally with syntax representations and is easy to generate and link with. Since the leaf nodes of the syntax tree are all tokens and are the *only* token nodes, the links between the representation are always to whole units or collections of whole units. This aids in consistency maintenance and incremental updates by mapping a change in one representation into the minimum number of changed units in the other. Since the text consists of both token strings and token separator strings, changes in the text which do *not* affect the parse are also easily identified. Finally, the display of the syntax tree can be focused on the non-textual aspects because it is easy to identify and highlight for the user that text which corresponds to a particular syntax unit.

The syntactic model is represented as a tree of syntax nodes; the root node corresponds to the program text stream to be edited and the leaf nodes are the tokens of the edited language, in our case Ada. There is no restriction placed on the number of children any node may have, except as may be defined in the edited language. We guarantee in our representation that all tokens end up as leaf nodes in the tree and all leaf nodes are tokens. Each node has a list of text segments on which it depends, generated by propagation up the parse tree. Every leaf node has exactly one text segment, and every non leaf node has all the text segments of all its children. Each text segment has corresponding pointers back to every syntax node that depends on it. In addition, to make the syntax representation more accessible to humans, barren stalks are collapsed. Barren stalks are any non-leaf node that does not have two or more children. To preserve the most specific information, a barren stalk is replaced by its child. We have not empirically found an example where the user would want to see the display of a barren stalk, but we retain the information should the need ever arise.

3.2 The Parser-Developer

3.2.1 Motivation

In order to create these two models from a string of program text, we needed an augmented parser, which we built from a public domain LALR grammar by Herm Fischer of USC for accepting Ada. We also use the YACC/LEX compiler writing system, as supplied with the Berkeley 4.2 distribution of Unix. We could not simply run the grammar through the system and use the resulting parser as is, nor could we afford to add the needed enhancements by hand for several reasons. A parser created by such automatic systems is not designed for human comprehension and is difficult to read and understand, let alone modify. Also, the grammar we used was for an Ada acceptor--there were no provisions for outputting any kind of parse. Thus we decided to automate the construction of a parser for a language from an LALR accepting grammar for that language.

3.2.2 Theory

The parser-developer is the result of our research into the automatic construction of parsers from accepting grammars. It allows us to build a parser quickly from a bare bones description of the syntax of a language. Although the PRL project is currently focused on Ada, this technique will not only allow us to keep the PRL up-to-date, it will also allow us to explore its use with other languages. With this tool, the PRL could be adapted to another language with little more than an accepting grammar for that language.

The parser-developer is closely related to compilers in that it was constructed with a compiler writing system and its input and output programs can be mapped one-to-one onto each other. It is *not* a compiler, however, in that the mapping does *not* preserve semantics--the output program specifies a more complex procedure than the input program.

The grammar for the Ada accepter is written in a language called YACC, which looks like a cross between Backus-Naur Form and the C programming language. This

grammar was originally intended to be compiled by YACC into the machine language Ada accepter which we needed. The parser-constructor reads in the YACC program for an acceptor and generates a similar YACC program for a parser. That is, the output of the parser-constructor is a YACC program which generates parse trees for the same language that the input YACC program accepts. The parse tree generators written by the parser-constructor already contain all the enhancements needed by our system so no manual intervention is necessary. The original Ada accepting grammar can now be run through the parser-constructor. The resulting YACC program, when compiled by YACC, is the parser that is used by the PRL.

3.2.3 Construction

We modified the compiler writing system available to us so that the parsers that are ultimately developed keep track of their position in the program text stream as they tokenize the stream. The manual which describes how to use YACC includes a YACC grammar written in YACC. We used that grammar as a starting point in building the parser-developer. From the YACC accepter grammar, we created a copying grammar, one that would exactly reproduce the input. This was tested on a number of YACC grammars, including itself, to verify that it was correctly recognizing and recreating the YACC programs. From there, we modified the YACC copying grammar to augment the code while copying, so that the new code could take advantage of the extensions to YACC to build parse trees. Note that the parser-developer is a YACC program which operates on an input YACC program and writes a new YACC program which bears a functional relationship with the input program.

3.3 Constructing the Syntactic Model

Once the idea of the parser-developer is realized, the details of how the resulting parsers (which are its outputs, given well formed inputs) behave is an independent issue. Our parsers work by following the internal state of the LALR engine created by YACC for a given input. Every explicit shift and explicit reduce transition causes a record of that transition to go to the output stream. Unfortunately for the cleanness of the implementation, YACC's output programs frequently do implicit shifts. Thus, some of the shift transitions that should be a part of a following reduce will be absent from the

output stream. However, since the internal state numbers are available at shift transitions, and a list of all the state numbers that should be popped and their exact order are available at reduce time, this deficiency is easily resolved.

The output of the parser is a stream of records of shift and reduce transitions. A LISP function within the PRL system reads these records and builds the actual parse tree data base from them. As it reads the stream, it keeps two stacks. The first one is a stack of syntax nodes which form the tree. The second is a stack of the state numbers that represent the state the YACC-base parser was in for that transition. The two stacks are always run completely in synchronization. Shift transitions push a new item onto each stack. Reduce transitions pop a number of items off each stack and push one new item onto each as well. The syntax nodes popped off the stack are the children of the newly created reduce node. When the process is finished, the top of the stack is the root of the parse tree. The deficiency mentioned in the previous paragraph is resolved at reduce time. The reduce record contains a "pop" list of states expected on the stack in the order they are expected to appear. The tree builder iterates down this list. Whenever the state at the top of the dual stacks matches the current state in the pop list, it is accumulated as a child of the new syntax node being created. Otherwise, the entry in the pop list is ignored. The pruning of null transitions, which the YACC-based parser insert as non token leaf nodes, and the collapsing of barren stalks occur at this time as well.

The shift transition record for tokens also include the position within the program text stream that the token starts and end on, as well as the value of the token. Because of the nature of LALR parsers, these always occur in the same order as they do in the text file. So, while the parse tree is being build, so is a list of text segments to accommodate the tokens. As this list is built, the text segments in it are bidirectionally linked to the syntax tree. Once the parse tree is finished, the list of text segments, which is already in the right order, is examined and filled in with text segments to hold the values of the token separators. The program text stream is then read in order to fill in the missing strings. Once that is done, the two representations are complete and properly linked, and ready for use by the PRL query handler.

3.4 Design of the Semantic Model

The semantic model is currently in a design phase. An outline of our design considerations follows.

The construction of the semantic model incorporates three phases:

1. Explicit transcription into the research language (LISP),
2. Minimization of the representational form, and
3. Orthogonalization of data and control.

The conversion process is expected to involve several intermediate steps and several intermediate representations. Although the final conversion process will not need every step, we know that it is important for design to make small incremental changes to the representation, so that we can build and debug modularly.

The sequence of representation languages used in the construction of the semantic form is:

1. Textual Ada: the syntactic and textual models of the source program.
2. Textual LISP: a literal mapping of the syntactic and textual models from Ada to LISP.
3. Elementary LISP: macro conversion of special forms (such as LOOP) into the elementary lexicon of LISP.
4. Pure LISP: iterative forms become recursive forms; assignments become argument bindings within functions.
5. Pure Mathematical LISP: accumulation forms are absorbed into argument forms (This step may be unnecessary.).
6. LAMBDA calculus: binding environments are made explicit.
7. LOSP: The LOSP language is a formalism in which all data structures are expressed as descriptive atoms, and all control and Boolean structures are expressed as nestings of parentheses, thus achieving a syntactic separation of data from control without loss of either expressive power or functional performance.

LOSP is an assembly language for deductive computation. It provides a rapidly convergent, complete and consistent decision procedure for propositional calculus. By eliminating the redundancy in traditional logical notation, LOSP achieves a parsimony that makes it particularly well suited for automation. The theory of representation, transformation rules, semantics, and proof methods of LOSP are available from Advanced Information & Decision Systems.

For a full description of the mathematics underlying LOSP, see Spencer-Brown's seminal mathematical text, LAWS OF FORM. We will not attempt a description of this work here.[1]

An Example of the EMP Conversion Process:

The following example illustrates the steps of conversion from Ada code to the EPM representation.

The construction process requires these explicit translation steps:

- EXPLICIT TRANSCRIPTION:
  - Literal Ada to LISP conversion
  - Make implicit structure explicit
- MINIMIZATION:
  - Literal iterative to recursive conversion
  - Remove invariant structures
  - Remove accumulation structures
- ORTHOGONALIZATION:
  - Literal conversion to LAMBDA form
  - Separate control from data by LOSP conversion.

[1]The inclusion of LOSP into the code for the EPM is a tentative design decision at this time, since LOSP is copyright protected software.

In the following example, the Ada form at Step 0 is the textual representation in the EPM. The syntactic parse tree that is used to convert ADA syntax to LISP syntax at Step 1 is the syntactic representation in the EPM. The remaining steps describe the construction of the semantic representation which is in the representation language LOSP at Step 7.

STEP 0: The ADA Source Code Example:

```
PROCEDURE TESTADA (X: in integer) is
  pragma MAIN;
  I, B:  integer;
    begin
      B := 0;
      for I in 1 .. X loop
        if I < 5 then
          B := B + I;
        else
          B := B - I;
        end if;
      end loop;
    end TESTADA;
```

STEP 1: Literal translation into LISP:

```
(defun LADA1 (X)
  (and (integer X)
       (setq B 0)
       (loop for I from 1 to X
         if (< I 5)
           do (setq B (+ B I))
         else
           do (setq B (- B I))
         finally (return B)
         ) ) )
```

STEP 2: Explicit LISP form:

```
(defun LADA2 (X)
  (and (integer X)
       (do ((I 1 (add1 I))
            (B 0) )
           ((or (< X 1) (> I X)) B)
           (if (< I 5)
               (setq B (+ B I))
               (setq B (- B I))
               ) ) ) )
```

STEP 3: Literal Recursive representation:

```
(defun LADA3 (X)
  (and (integer X)
       (LADA3-AUX X 1 0)
       ) )
(defun LADA3-AUX (X I B)
  (cond ((or (< X 1) (> I X)) B)
        ((< I 5) (LADA3-AUX X (add1 I) (+ B I)))
        (t (LADA3-AUX X (add1 I) (- B I)))
        ) )
```

STEP 4: Remove Iteration variables:

```
(defun LADA4 (X)
  (and (integer X)
       (LADA4-AUX X 0)
       ) )
(defun LADA4-AUX (X B)
  (cond ((< X 1) 0)
        ((< X 5) (LADA4-AUX (sub1 X) (+ B X)))
        (t (LADA4-AUX (sub1 X) (- B X)))
        ) )
```

STEP 5: Remove Accumulation variables:

```
(defun LADA5 (X)
  (and (integer X)
       (cond ((< X 1) 0)
             ((< X 5) (+ (LADA5 (sub1 X)) X))
             (t (- (LADA5 (sub1 X)) X))
             ) ) )
```

STEP 6: Lambda form:

```
(bind LADA6
  (lambda (X)
    (and (integer X)
         (cond ((< X 1) 0)
               ((< X 5) (+ (LADA5 (sub1 X)) X))
               (t (- (LADA5 (sub1 X)) X))
               ) ) ) )
```

STEP 7: LOSP Removes control functions:

```
(bond LADA7
  (lombda (X)
    ( ((integer X))
      (((< X 1)) 0)
      ( (< X 1)  ((((< X 5)) (+ (LADA7 (sub1 X)) X))
                  ( (< X 5)  (- (LADA7 (sub1 X)) X))
                  )) ) ) )
```

In the following Chapter, we discuss design considerations for search and retrieval of information from the EPM.

4. Efficient Retrieval for the EPM

4.1 Searching in the Extended Program Model

Since the purpose of the Program Reference Language is to provide a way to reference programs, one of the key features of a PRL system must be a search facility. The design of the search theory is based on the proposed PRL Picture Language, a pictorial representation that provides the ability to specify high-level search requests. Using this as a foundation allows the study of the development of basic searching techniques. Even if the PRL Picture Language is not the final query mechanism used in the EPM, the intermediate language into which the Picture Language will be translated for access into the EPM will be similar in nature to the intermediate language that is eventually used in the system. The remainder of this chapter assumes the PRL Picture Language to be the user interface into the EPM.

The PRL research effort is oriented towards providing better methods for program comprehension. Program comprehension becomes of particular importance when dealing with large programs (which may exceed the limitations of what one person can understand). Thus, one of the goals of the PRL research is to provide techniques that will work for large programs.

In dealing with large program search spaces, there are two critical issues: precision and efficiency. The term "precision" refers to the percentage of retrieved items that are actually of interest. As the size of the retrieval space increases, precision becomes more critical. In small search spaces, a small number of false hits are only a minor annoyance to the user; however, if the search space is increased by several orders of magnitude, the corresponding increase in false hits becomes unacceptable.

The PRL inherently simplifies the process of making requests more precise. By allowing the formulation of requests on the basis of various representations (i.e., text, syntax, flow, and/or cliches), requests are easier to formulate *and* they are more specific (and constrained) than simple (text-only) requests.

When the size of the retrieval space increases, the issue of efficiency also becomes more critical. In dealing with a single module, it may be acceptable for the search process to be relatively slow; in dealing with hundreds or even thousands of modules, the time is multipled accordingly, and quickly becomes unacceptable in an interactive programming environment. This Chapter describes how PRL-based searching can be performed efficiently; using the techniques described here, PRL-based search may even be more efficient than conventional searching.

4.2 Intra-Module Query Processing

To explain the PRL search process, it is first necessary to understand the internal representation of programs in the Extended Program Model (EPM). The EPM can be though of as database for storing programs. Its key feature is that it stores programs in a variety of different representations.

The EPM is segmented along two axes (see Figure 4-1). Each module is represented independently in the database (inter-module relationships are represented separately). Within a module, each representation is stored separately; however, there are links between the representations, which provide the basis for converting between representations. Modules provide a natural division from the programming viewpoint; similarly, they provide a natural division from the retrieval perspective. Thus, PRL supports two types of searching: searching within modules and searching between modules. The intra-module search process is the basis for the inter-module search process.

The basic process in retrieval is the application of a PRL query against a single program module. Much of the sophistication of the PRL is based on this capability, which manipulates the multiple representations supported by the EPM in order to handle the full expressiveness of the PRL. The single module retrieval process is currently being researched, and is described here only briefly; for the purpose of explaining inter-module search, the intra-module retrieval process can be treated as a black box.

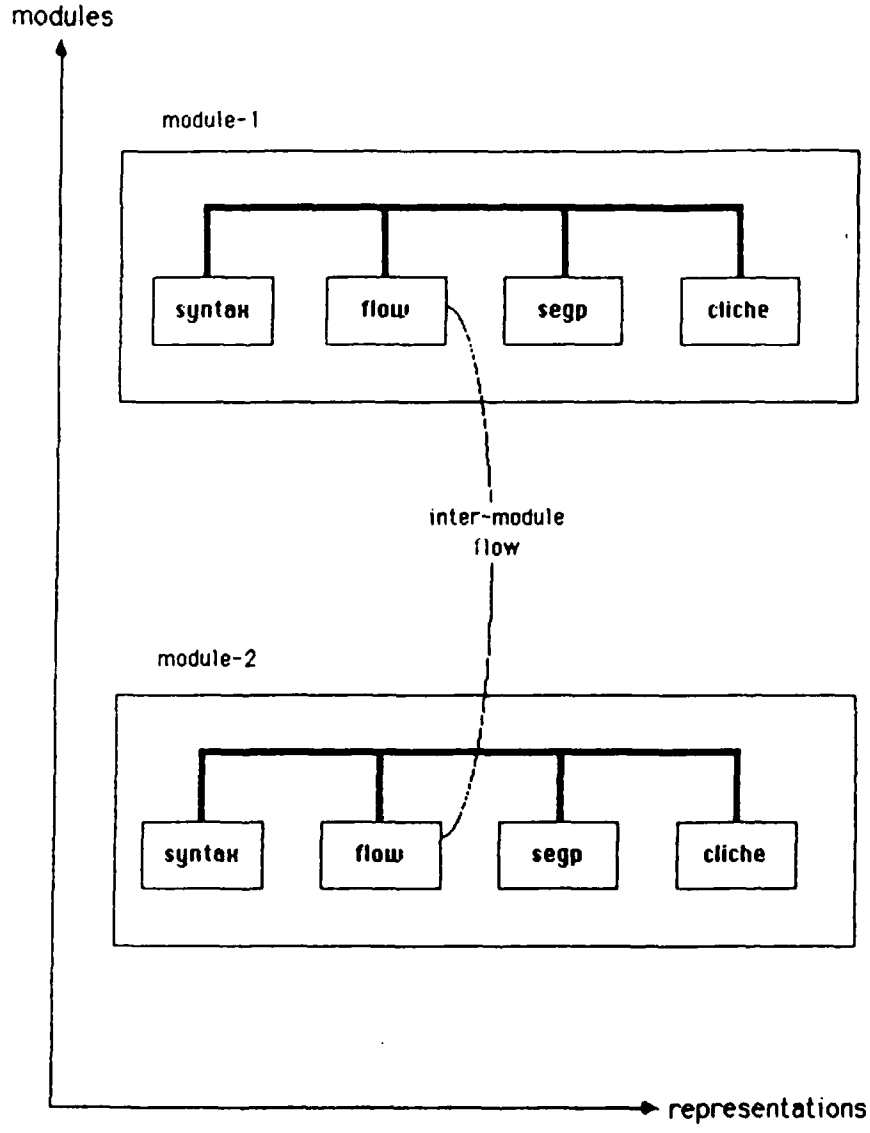


Figure 4-1: Structure of the EPM

As currently envisioned, the intra-module search mechanism will handle requests based on the PRL Picture Language. Search requests are limited in their complexity since the PRL/PL restricts the structure and vocabulary of queries. Moreover, the EPM makes several simplifying assumptions to make search easier. As a result, the cost of intra-module searches should be roughly proportional to the size of the module.

A PRL/PL query specifies objects and their relationships; it may also contain boolean connectives. The search mechanism supports the three basic PRL/PL relationships:

- *containment:* structural nesting of objects ("A" is inside "B")
- *precedence:* temporal or spatial ordering of objects ("A" comes before "B")
- *attribute:* characteristics of objects ("A" has a "B")

Containment- and precedence-based searching is similar. All objects to be searched are converted into a common representation (e.g., containment requires all objects to be converted to a syntactic representation); after conversion, a tree traversal is sufficient to identify target objects.

Searching based on attributes requires search through the class hierarchy for the appropriate attribute. If the attribute is found, representation conversions and/or computations may be required.

Search may require objects to be converted to different representations. The EPM currently provides one-to-one mappings between representations, simplifying the conversion process. A later version will support fuzzy and one-to-many relationships, possibly requiring searching and computation to do conversion (and thereby increasing the cost of intra-module searching).

4.3 Efficient Inter-Module Query Processing

Inter-module retrieval can be done by repeatedly applying the intra-module search mechanism to each module in the system and concatenating the results. This basic process will be quite inefficient, and not all requests can be satisfied by intra-module searching.

Performing inter-module search by repeated application of the intra-module search process over the entire set of modules is referred to as the *naive* model. It represents an upper bound on the amount of work that a search process must do -- i.e., search everything. In a large application, the naive model becomes prohibitively expensive (and unusable in real-time), since the per module search costs are multiplied by the number of modules.

There are a variety of methods that might be used to reduce the search space. The technique described here is based on indexing, a well-known technique that seems especially suitable for the PRL. Our previous experience with applying indexing techniques indicates that speed improvements (over naive methods) of several orders of magnitude are possible.

4.3.1 Indexing

An index can be thought of as a mapping between the terms of a retrieval request and the target space. Given a request term, an index can be used to locate that term in time proportional to the size of the index, instead of the size of the entire search space. There are two basic types of indices. A *direct* index provides a mapping from modules to terms. It answers questions of the form *"Where does term-1 appear in module-1?"* An *inverted* index provides a mapping from terms to modules. It answers questions of the form *"Which modules contain an term-1?"* Figure 4-2 presents an example.

The EPM will have several inverted indices, one for each program representation (Figure 4-3). Each inverted index maps from an object (in that representation) to a module. For example, in the syntax inverted index, there is a list of all modules containing *loops*, a list of all modules containing *assignments*, etc.

The indices are used to *filter* out modules that cannot possible satisfy a query, before the intra-module search process is invoked on that module. Since filtering is (by design) a much simpler process than searching, considerable speedup can be achieved by eliminating modules from consideration. However, the reduction of the search space depends on the particular terms in a query. If, for example, a query contains terms that are contained in few of the modules, then many modules will be eliminated, and a

term	modules
case-statement	module-4
goto-statement	----
if-statement	module-1 module-2 module-3 module-5
loop-statement	module-1 module-3 module-5

Figure 4-2: A Simple Inverted Index

significant speedup will be realized; on the other hand, if the terms are common, the filtering process may actually cause the retrieval process to take longer than a naive query process.

The technique used for filtering modules can be thought of as a stripped down version of intra-module query evaluation algorithm. It converts the query tree into an AND/OR tree. The AND/OR tree represents the minimal requirements a module must satisfy for the full query to succeed.

The AND/OR tree is constructed by converting all relations (containment, precedence, negation, quantification) into AND or OR relationships. If the query requires the presence of some object (e.g., the query *"find an A which contains a B"* requires both A and B), then the conjunction of A and B is added to the AND/CR tree. If an object is not required (e.g., the query *"find an A which does not contain a B"* does not require B), then it is omitted from the tree. Figure 4-4 shows an example of a PRL/PL query, its translation to tree form, and the resulting AND/OR tree.

To determine if a module is a candidate for full searching, the AND/OR tree is evaluated with respect to the indices for that module. Only modules that evaluate to *true* are candidates for the full query evaluation; the other modules are missing

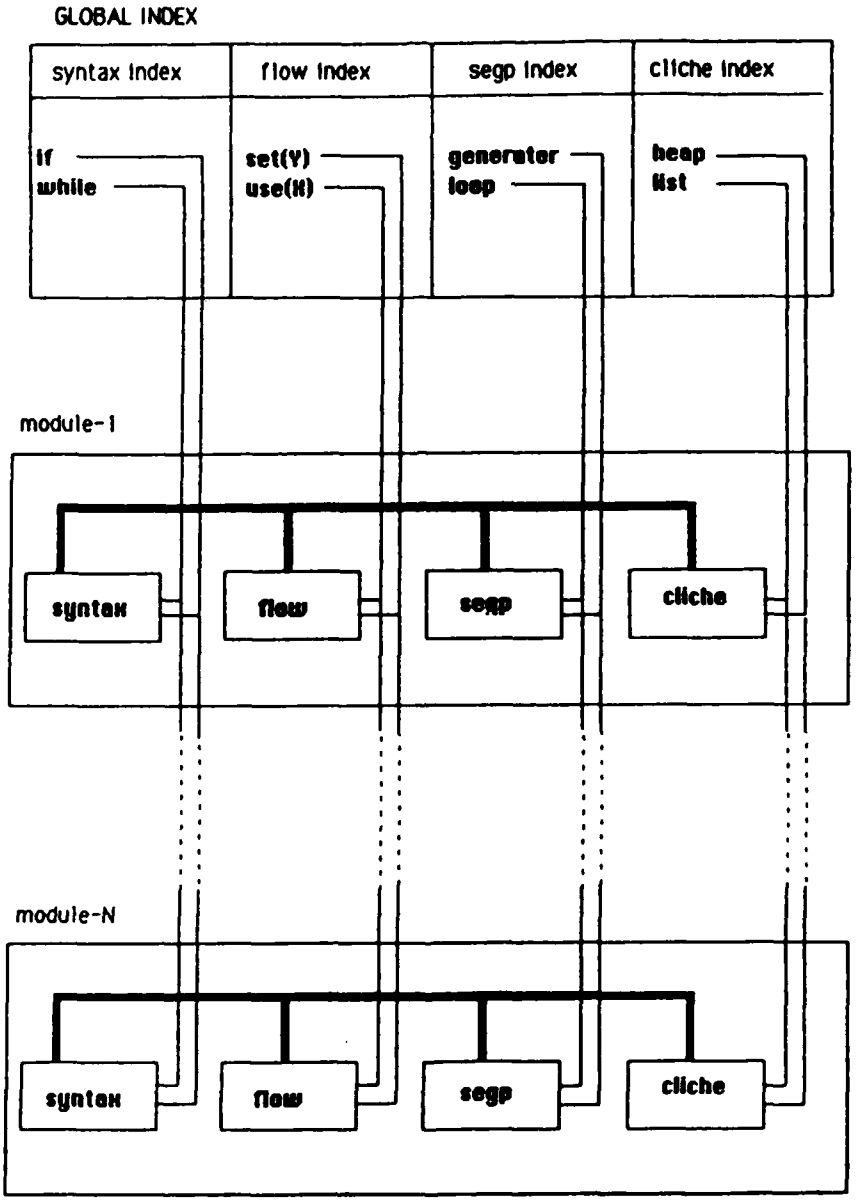


Figure 4-3: Inverted Indices in the EPM

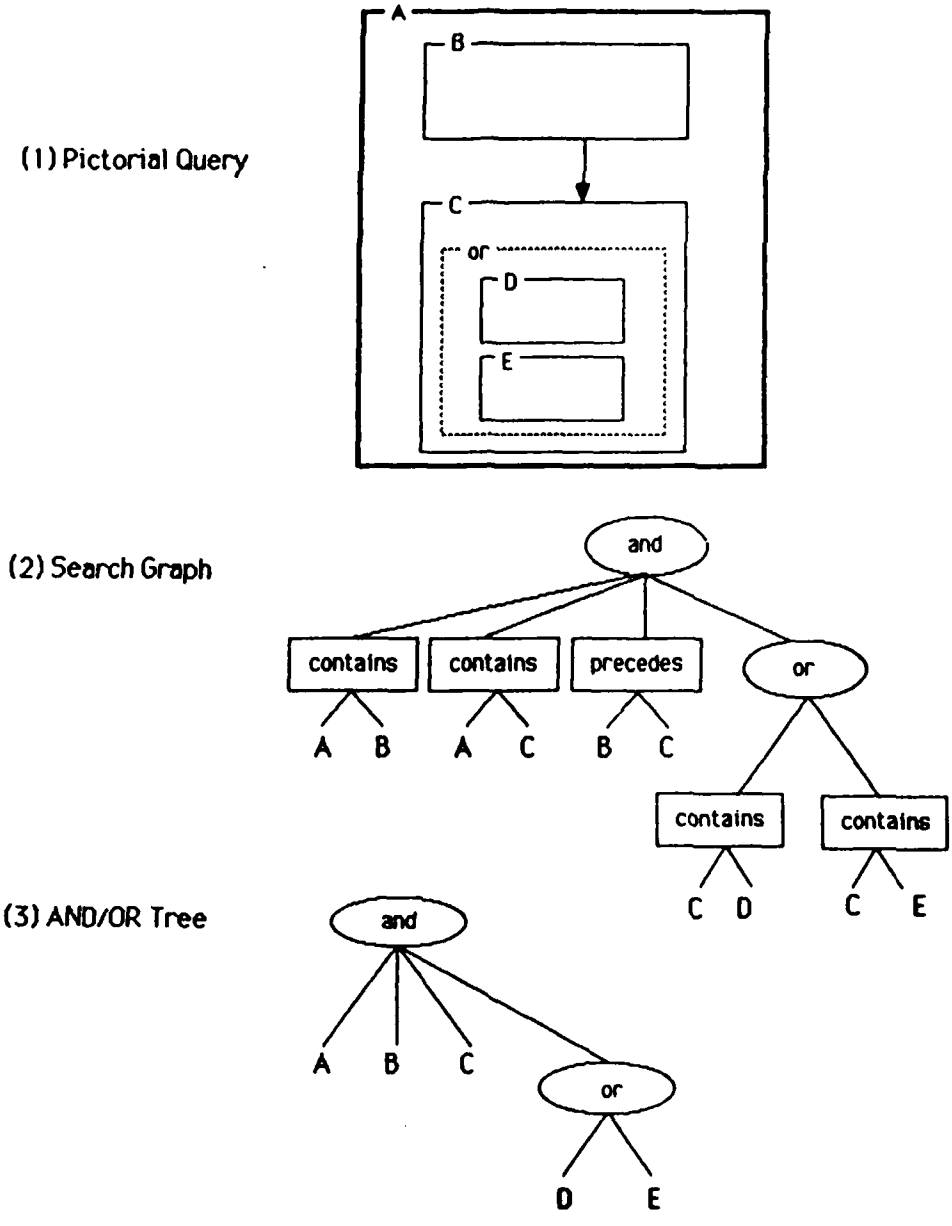


Figure 4-4: Mapping from a PRL/PL Query to an AND/OR Tree

necessary components. Thus, the filtering algorithm can be thought of as "structureless" query processing -- all of the components of the original query must be present, but their relationships are irrelevant.

Unless the filtering process actually eliminates some modules, the cost in this case would actually be higher than the naive case. In a large system, it is expected that filtering will in fact reduce the number of modules. A technique for computing the break even point for the filtering process (i.e., the number of modules that have to be eliminated for the filtering to be cost effective) remains to be determined.

4.3.2 Filtering

The filtering process operates on the inverted indices (and not the modules themselves). The process can be performed either sequentially (evaluating the AND/OR tree once for each module) or in parallel (evaluating the AND/OR tree once for all modules). From an efficiency viewpoint, there are different advantages to both; selecting a particular method might best be done empirically.

The parallel evaluation technique evaluates all modules at each node in the AND/OR tree. A bit vector is used to represent the modules, with one bit indicating the status of each module. At the end of the evaluation, the bit vector indicates which modules passed through the filter. For each operation in the AND/OR tree, if there are M modules in the database, M boolean operations are performed on the bit vector. Parallel evaluation allows no shortcuts: if there are N operations in the tree, O(M * N) boolean operations must be performed. However, bit vectors are compact, and can be executed quite efficiently by most computers, so the total overhead is reasonable.

The sequential evaluation technique accesses the tree once for each module (and accessing the set of indices about that often). The sequential process permits short-circuit boolean evaluation, allowing modules to be eliminated without necessarily evaluating the entire tree. It can also make use of frequency information to reorder the tree into a more optimal form, allowing early evaluation of those indexed items with the highest discrimination ability. Thus, sequential evaluation is much more sensitive to the data; determining the associated cost will require experimentation.

It might be possible to perform a pre-filter process, to reduce the cost of the filtering process itself. The pre-filter process could be an even more reduced version of the filtering process, or it could be based on statistical information about previous filterings.

4.3.3 Minimizing Index Size

Non-indexed retrieval schemes have the advantage of requiring no extra overhead to support retrieval; retrieval can be done directly on the modules themselves. Indexing schemes have the overhead of creating and maintaining indices. To prevent the cost of the overhead from exceeding the savings achieved by indexing, there are several ways of reducing indexing overhead.

4.3.3.1 Simplifying the Index

In our indexed retrieval model, only a simple indexing structure is needed. Since the index is used to answer true/false questions about each module, the index needs only an indication of whether an object is contained in a module. It is not necessary to specify where in the module it appears.

This greatly simplifies index maintenance, since it is only necessary to update the index when a new object is added to a module or when the last of an object is removed. It also reduces the size of the index, since it is only necessary to store at most one indicator per object/module pair.

Moreover, index maintenance in an environment like the EPM can be done incrementally, as programs are modified. This can even be done in the background, invisible to the user.

4.3.3.2 Limiting the Index

The vocabulary of the PRL contains all objects represented by the EPM. A simplified indexing scheme would require all terms in the vocabulary to be indexed. However, the indexing scheme proposed here does not require that all terms be indexed; doing so might actually be detrimental to efficiency.

The purpose of the index is to filter out modules that cannot possibly contain the sought objects. Thus, an object that occurs in every module is useless with respect to indexing, since it will never cause a module to be eliminated. An object that appears in most modules would also be less useful as an index term, since it too would be a poor filter. For an object occurring with great frequency, it may be less work to assume that the object is in all modules than to actually perform filtering based on that term.

On the other hand, objects that occur with too low a frequency may not be worth the expense to index, especially if there are a large number of these objects, since these objects would be rarely specified in search requests. The best example of this is in at the textual level, where it is infeasible to maintain an index of all possible substrings.

The decision about what to index and what not to index can be made at design time, when the indexing mechanisms are being constructed, or at run time, when frequency information is available. The best strategy might be to make these decisions at both times. For example, it can be decided a priori that maintaining indices for the textual representation is unlikely to be useful; however, deciding which syntactic features should be indexed would be best determined a posteriori, using frequency information.

4.4 Joining

The search model described earlier was based on the assumption that all queries could be satisfied by examining a single module at a time. In fact, this limitation prevents the general use of meta-variables, which provide a means for joining arbitrary components in the program space. However, our basic model can be extended to handle meta-variables.

The algorithm for processing a query with meta-variables operates by decomposing the query into subqueries without meta-variables. All of the subqueries are then processed as regular queries, using the previously described indexed retrieval algorithm. Then, the results are joined together, which returns the final result.

Queries are *decomposed* in two steps:

1. *find lowest common ancestors:* For every pair of meta-variables, locate the lowest common ancestor in the tree; this node is referred to as the *join node* (Figure 4-5-1). A join node may actually join any number of meta-variables.

2. *break off branches:* At each join node, break off all branches that contain meta-variables (Figure 4-5-2). The resulting branches (including what is left of the original tree, if anything) are the new joinless queries (i.e., no branch has more than one meta-variable).

This set of subqueries is then processed in the standard fashion (i.e., including filtering). Subqueries containing single meta-variables are handled by letting the meta-variables match anything satisfying specified constraints.

Each subquery will have produced a set of values as a result. These results are then *combined* as follows:

1. *rebuild tree:* Associate the resulting set of values with the top node of each branch, and put the entire tree back together (Figure 4-5-3).

2. *join results:* The intermediate results belonging to the join nodes can now be computed by performing an operation on all the result sets of the children (Figure 4-5-4). The specific operation depends on the type of the join node. Specifics are presented in Figure 4-6.

The *equi-join* and *Cartesian product* operators are relational algebra operators. Each branch of the divided query produces a set of tuples as a result. Equi-join merges two branches by combining pairs of tuples (one tuple from each branch) that have matching meta-variables. Cartesian product merges two branches by producing all possible pairwise combinations of tuples. Operations for other join nodes are the functional composition of the node relation with Cartesian product.

Joining can be an expensive operation, especially when the number of objects to be joined is large, as can happen when the join is applied across the entire program space. If it can be determined that a join operation is constrained to occur within a single module, then the join can be performed on a per-module basis, as part of the intra-module search process.

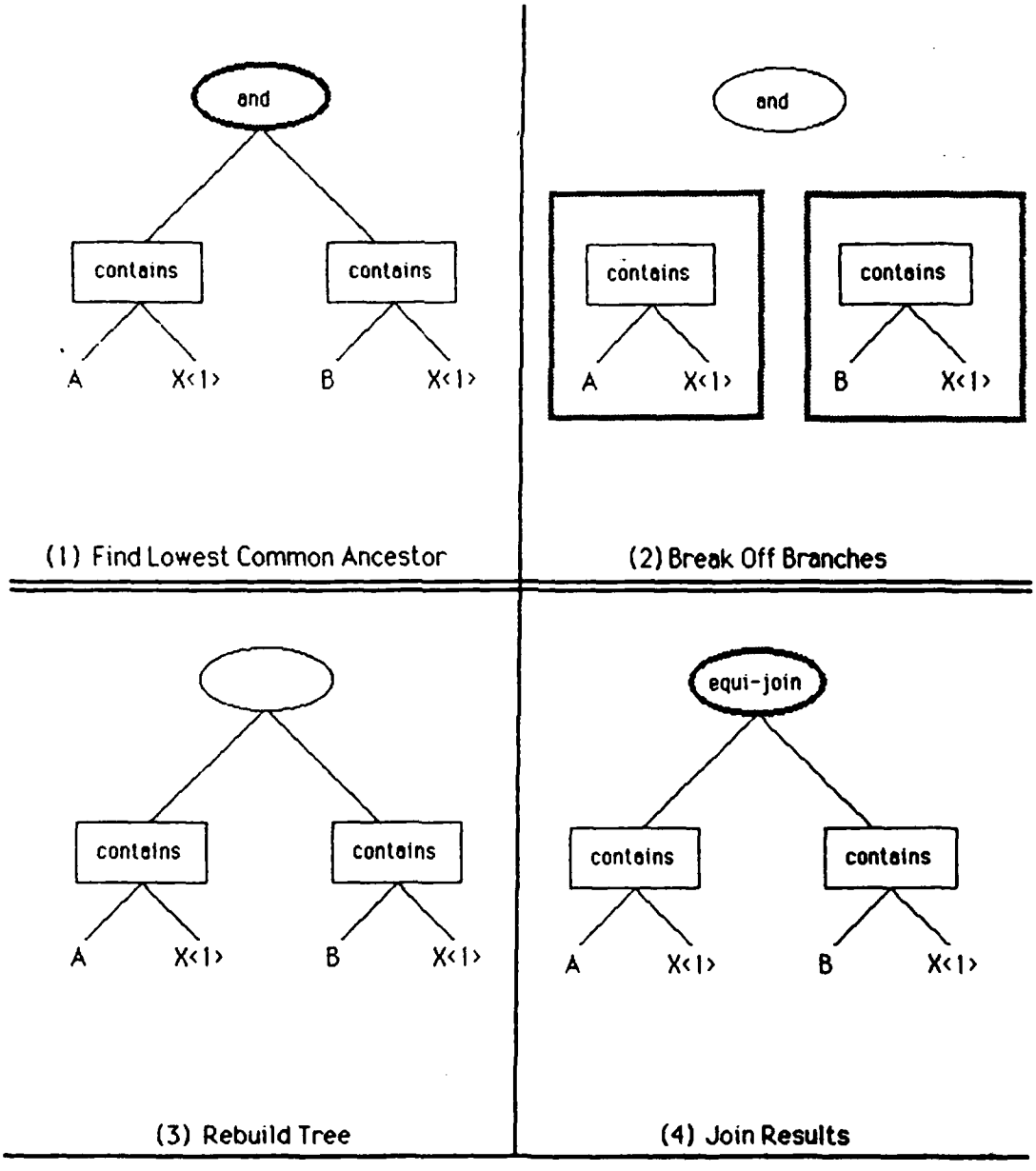


Figure 4-5: Handling Joins in Searches

type of join node	*operation*
and	equi-join
or	Cartesian product
relation	*relation* • Cartesian product

Figure 4-6: Joining Operations

4.5 Summary

Due to the sophistication of the PRL and EPM, we originally expected PRL-based search to be a slow and inefficient process; we now believe that the PRL may be able provide efficient searching across large programs. Moreover, because the EPM provides a natural and well-defined basis for indexing -- the multiple program representations -- it may turn out that PRL-based searching can be performed more efficiently than conventional searching.

5. The EPM Query Language

In this Chapter we define a formal query language to access the representational structure of the EPM. Queries of the textual representation are available in standard string matching editors such as EMACS. To distinguish between textual and other requests, the user must place a single quote (similar to LISP) in front of a textual atom, or a double quote around a textual string. For example:

Find the ':=

will create a text search for the token string ":=".

Find the ":="

will have the same effect. However,

Find the :=

will search the semantic database for the act of assignment.

5.1 Design of the Intermediate Query Language

The intermediate query language is based on LOGLISP (Robinson, 1981) and on the Prolog Database System ILEX (Li, 1984).

The syntax of the intermediate query language is LISP-like:

`(<quantifier> <answer variables> <specification>)`

The QUANTIFIER field may consist of these options:

- ALL, which returns all occurrences in the current context.
- THE, which returns the one specific occurrence.
- ANY K, which returns the first K occurrences.
- COUNT, which returns the number of occurrences.

The ANSWER VARIABLE field is a list of dummy variables mentioned in the specification for matching.

The SPECIFICATION field may be any Relational Calculus form, with atoms drawn from the permitted identifiers, and relations drawn from the permitted relations,

described below. The current set of descriptive identifiers is: CONSTANT, VARIABLE, FUNCTION, LOOP, ASSIGN, and IF.

For the scope of this research, we have limited the current context to a single PROCEDURE. In the future, this will be extended to include queries that address relations between procedures.

At this stage of development, we allow the following relations in the specification. Note that relations may be interpreted as textual queries if the arguments are quoted, or as semantic queries, if the arguments are not quoted.

Contains["form1", "form2"]:
The structure represented lexically as FORM2 is contained within the syntactic scope of FORM1.

Contains[form1, form2]:
Error: containment is not a semantic concept.

Follows["form1", "form2"]:
The lexical structure FORM1 follows (in space) the lexical structure FORM2, within the current context.

Follows[form1, form2]:
The invocation or evaluation of FORM1 follows (in time) the invocation of FORM2.

Uses[form1, form2]:
The invocation of FORM2 occurs within the scope of the invocation of FORM1.

Sets["form1", "form2"]:
Within the syntactic scope of FORM1, the lexical FORM2 is followed by the lexical form of assignment, which may be ":=" or it may be active use of a LAMBDA-bound FORM2.

Sets[form1, form2]:
The structure represented by FORM2 changes value within the scope of invocation of FORM1.

The definition and refinement of permissible relations is an active area of research in this project.

5.2 How the Intermediate Query Language Will Work

The user may, of course, bypass the Picture Language interface or other forms of interface language, and express queries directly in the intermediate form.

Propositional queries of a semantic nature are satisfied by converting the specification to an EPM pattern and pattern matching. Syntactic queries are pattern-matched against the syntactic database.

We have not yet formed operational definitions of the relational queries, or of the predicates that will form the basis of the query language capabilities.

Some examples of possible queries are included in Appendix A. Each query is presented is three different representation languages: predicate calculus, LISP pseudocode, and the formal Picture Language which is isomorphic with LOSP.

6. The Formal Picture Language

A major thrust of the PRL Project is to develop a query language and an accompanying interface that simplifies the submission of program queries to the EPM. Last year's annual report, entitled Design of a Pictorial Program Reference Language (AI&DS Final Report TR-1014-4), described a pictorial language that is intuitive and easy to use. For simple requests, this pictorial language permitted the user to access the EPM database by creating two-dimensional representations of logical structures that contained syntactic identifiers.

The former picture language suffered from a serious defect: it was inconsistent. This defect is intolerable for these reasons:

1. Complex requests were ambiguous, and could not be translated into consistent queries. Representations did not map one-to-one onto specifications.

2. The user would be reinforced to think about program specifications that could not be expressed in a formal language. The user's model of the system would be undermined by inconsistency.

3. The language was incomplete, forcing additional relations. For example, both CONTAINS and IS-CONTAINED-BY were necessary.

4. Some operators, such as negation, produced counter-intuitive results. Object tokens became confused with logical values.

Thus, our research direction focused on the formalization of the Picture Language.

6.1 Formal Specification of the Picture Language

There are two simple, formal, pictorial representations of logical structure, which are dual to each other. Table 6-1 specifies our formal pictorial logic. Parentheses should be read as geometrically surrounding figures, such as circles.

```
a                       ==>    a

a AND b                 ==>    a  b

NOT a                   ==>    (a)

a OR b                  ==>    ( (a)  (b) )

IF a THEN b             ==>    ( a (b) )
```

Figure 6-1: The Formal Picture Language

CONJUNCTION is achieved by placing tokens within the same representational space. No token of conjunction is used. That is, conjuction is implicit in the space of reference.

NEGATION is expressed by the encapsulation of a variable, surrounding it by a geometric figure, such as a circle, or in a linear representation, by parentheses.

OR and IF can be seen as logical/pictorial compositions of their definitions in terms of AND and NOT. Specifically:

```
a OR b  =  NOT ( (NOT a) AND (NOT b) )
        =      ( (    a)     (    b) )   =   ((a)(b))
```

Since AND is implicit in the PL representation, and since NOT is redundant with the parentheses, the above expression simplifies to a form that contains no explicit conditional or control functions.

Similarily, IF is constructed as follows:

```
IF a THEN b  =  NOT ( a AND (NOT b))
             =      ( a     (    b))     =   ( a (b))
```

This very simple transformation from the representation of logical connectives as tokens to their representation as figures has two remarkable properties:

1. The representation requires only one (super-dimensional) token, that of the parentheses or the circle.

2. The representation is isomorphic with Propositional Calculus.

In this representation, the logical forms of an expression is stored in a higher dimensional space than are the variable forms. The consequence is a single operator logic than is not representationally explosive (as is, for instance, the Sheffer stroke). Since geometric representations do not interact with linear representations, the expression of control as parentheses is independent of the expression of data as variables.

6.2 Other Representation Issues

The primary relation that the Picture Language is intended to embody is that of CONTAINMENT. To achieve this pictorially, we overloaded the representation of IF to also mean CONTAINS. Thus

```
( a (b) )
```

is the representation for both "IF a THEN b" and "a CONTAINS b". The intuitive mapping is analogous to the mapping between "b IS-A-SUBSET-OF a" and the Venn Diagram expressing this relation, abstractly: (a (b)).

To alleviate the necessity for inverse relational expressions (CONTAINS and IS-CONTAINED-BY), we highlight the expression intended as the focus of search. Thus:

```
( a ( *b* ))
```

specifies "Find the Bs contained-by A." Conversely,

```
( *a* ( b ))
```

specifies "Find the As containing B."

To summarize our representation techniques for the Picture Language:

1. Boolean operators are expressed by two dimensional figures surrounding tokens.

2. CONJUNCTION is implicit when forms share the same representative space.

3. NEGATION is represented by containment in a geometrical figure.

4. Other logical connectives are defined in terms of AND and NOT.

5. The CONTAINMENT relation is overloaded on the representation of IF.

6. The focus of search is highlighted.

6.3 Examples

Our previous final report (*Design of a Pictorial Program Reference Language*, AI&DS TR-1014-4) studied 18 query requests from the perspective of the informal picture language. Appendix A re-examines these 18 requests from the perspective of the formal picture language. This Appendix includes

- A formal specification of each query expressed in Relational Logic.
- A search program that achieves the request, written in an abstract version of LISP.
- The evolutionary development of the EPM search form of the query. The reader can trace a specification of the algorithm that translates an EPM query into a query expressed in Relational Logic.
- An abstract representation of the output of the search procedure.

Appendix A also contains working comments and notes that were developed during this translation exercise.

7. The User-PRL Interface

The intent of the PRL is to provide the user with a friendly interface with which to make requests about program structure. The interface is currently being fully redesigned. We expect the implementation of the interface described in this Chapter to be a two-year project.

7.1 Design Considerations

Our design focus has been to identify those conventions that both maintain a mapping onto the formal language and encourage an ease of expression of the informal intentions of the user. Several ideas need to be evaluated:

1. MIXED REPRESENTATIONS: Users should be able to freely mix pictorial elements of the PRL with string identifiers from written English. Further, users should be able to construct requests using mixed representations: part pictorial, part informal English, part formal logic. The request parser will convert these to expressions in a single formal language, either logic, code, or pictorial.

2. AUTOMATIC SIMPLIFICATION: Since the EPM form is a canonical representation of the source code, the system will be able to identify awkward constructs and redundant logical structure. The user could be notified of:

   clumsy control structure and how to improve it,
   clumsy and redundant variables and how to minimize memory,
   clumsy and confusing coding styles and how to clarify them.

3. INSTRUCTABILITY: For the request parser to understand informal (and even inconsistent) requests, it will need to be able to enter into a dialogue with the user about the intent of the request. The parser will default and/or coerce requests into a formal specification, notify the user of what has happened, and request conformation or clarification of the request. Additional techniques include maximal display of information and continual feedback.

4. FRIENDLINESS: We are developing a coordinated package of user-friendly display and interface options that inform the user of the interpretation that the PRL places on a request. These include:

   a. *Multiple parsing languages*: The user can see the request expressed in a variety of representations (for example: English, EPM, logic, search code).

b. *Multiple access techniques*: Request items can be typed, mouse-clicked, menu-selected, or defaulted.

c. *Dynamic parsing display*: The request is displayed in terms of both a formal language and the expected results. Requests are dynamically redisplayed when altered, and dynamically simplified when possible.

d. *Empathetic searching*: Using AI techniques, the interface can refer to a model of the intentions of the user, displaying queries and notifications whenever the request does not meet the modelled intents. For example, if a request returns 400 instances, the interface would ask the user if that was intended, or should the request be further constrained.

e. *Dynamic display of dynamic processes*: The user has the option to see the search process unfold dynamically instead of the traditional static display of request then results.

7.2 Interface Prototyper

7.2.1 Overview

In order to access the visual impact of the PRL, we have implemented a tool, called the Mock Engine, that permits the easy construction and manipulation of pictorial elements and program tokens. (Appendix B contains the documentation for using the Mock Engine.)

This tool evolved into an interface prototyping system that could be used for quickly developing the visual presentation that would be supplied to the user. Not only could this facility be used for the immediate need of the PRL work, but it could also be used for quickly developing many other user interface representations long before coding began; this would allow the designer to easily modify his ideas based on others' inputs without the problem of having to change the actual system based on their responses.

In addition to being able to study the display presentation, the system was also designed so that the process of a user sequencing through a set of commands could be emulated. The system developer can create a group of displays that would illustrate a set of user interactions. The displays can then be saved on a file to be read into the system and viewed in a movie-like fashion.

A supplementary capability is provided by the screen dump facility which allows the designer to print an image of a monochrome screen onto the laser printer. This allows design modification and critiquing to occur both on- and off-line, as well as providing for the production of slides from the black and white display.

7.2.2 Design

The interface prototyper consists of two basic components: a simple graphics editor with which to develop screen pictures and a mechanism for storing either one image or a sequence of images for later display. The graphics editor allows the user to place shapes on a screen and then manipulate these shapes through a series of predefined commands.

The shapes provided by the graphics editor include circles, rectangles, and arrows. Each of these objects can have his position changed and size modified. The rectangle handles more complex commands. It is actually a small editor window that allows the user to add text using the normal EMACS command sequences. Two other modifications the user can make to the rectangle shape alter its borders and background shade. The user can increase or decrease the size of the borders of a rectangle and have a gray-scale pattern as its background rather than the default white background.

The shapes are created and modified through menu selections from a command menu located along the right-hand side of the screen. The user would first select a shape to be generated, place the shape in the desired location, and then select from the existing commands to modify the shape.

Once the user is satisfied with a display, he can ask the system to save it as an individual unit on a file, or he can insert the display into a list of displays that the system keeps internally. Displays can be saved onto a file; the system will store enough information about the individual graph layouts to enable their recreation. The list of displays provide the interface prototyper's movie-like "animation" capability. With this feature, the user is able to serially display a succession of saved graphs. The user reads the list of displays back into the system and then uses the mouse button as a sequencer. Using the mouse in combination with input commands, the user can cycle through the

pictures, thus simulating a real interaction sequence with the system. This sequencing feature will be valuable in allowing critiquing of both the user interface display and the visual appearance of a user's session.

7.2.3 Use with PRL Picture Language

The Mock Engine was used to provide a visual presentation of the manner in which a user would develop a database query using the PRL Picture Language. Each step in the query building process was saved on the configuration list; once all steps had been developed and saved, the entire sequence was saved to a file. An example of the process used in developing the query *"Find all functions containing loops and if-statements"* will be presented in the following four Figures. Figure 7-1 displays the *FUNCTION* box; Figure 7-2 shows a *LOOP* contained within the *FUNCTION*; Figure 7-3 shows a *FUNCTION* box containing both a *LOOP* and a *IF-STATEMENT*; finally, in Figure 7-4 *FUNCTION* box is heavily bordered showing that all *functions* matching the query should be returned from the search.

The building of the query can then be simulated using the "configuration sequencing" capability of the Mock Engine. It is often far easier for a viewer to be able to understand and comment on a visual presentation than on strictly textual commentary.

All of the queries discussed in the PRL Picture Language chapter of last year's annual report, "Design of a Pictorial Program Reference Language" have been developed using the Mock Engine. Besides giving the user a pictorial view of the query development process, this work has also been helpful in showing the system designer's which queries could become tedious to develop due to the large number of mouse selections necessary for their creation. A system developer building an actual picture language query mechanism would be able to use this knowledge during development.

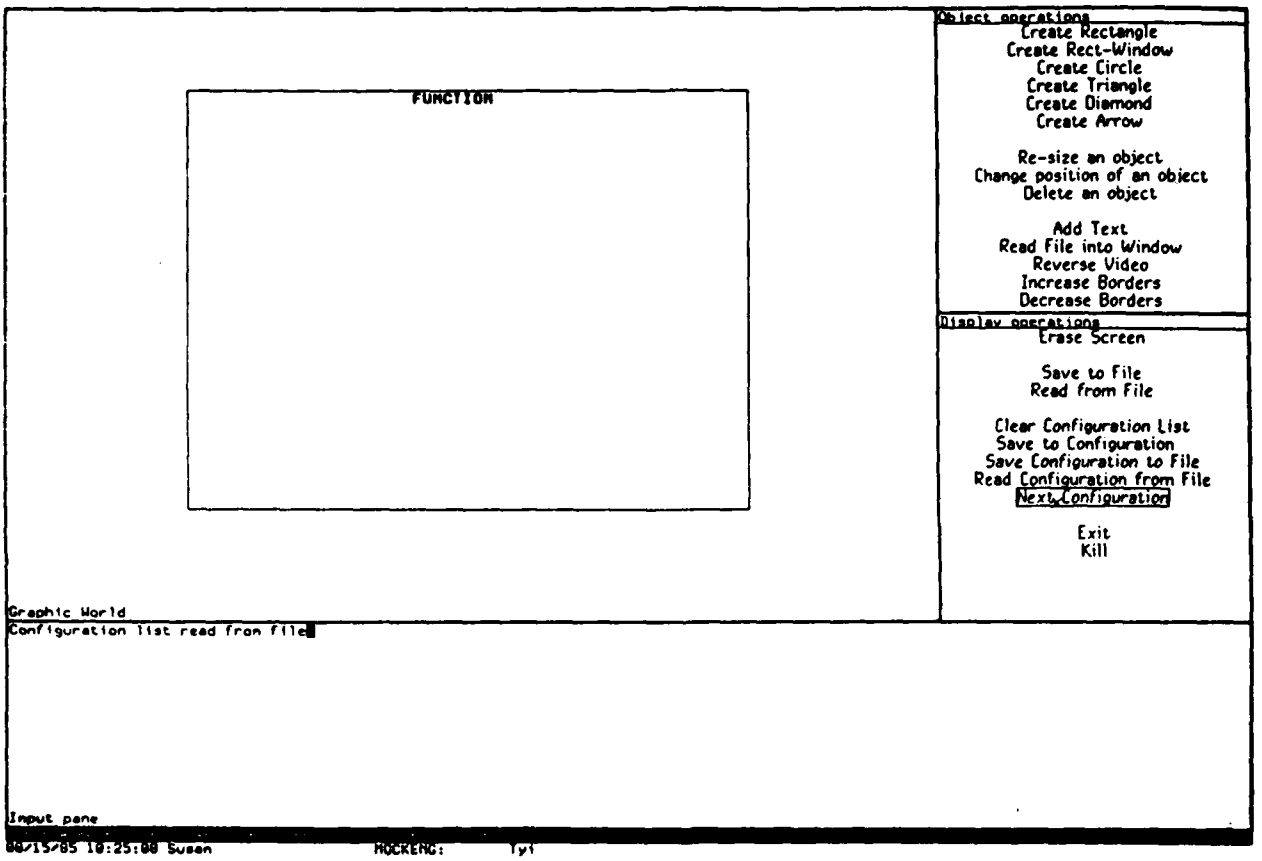


Figure 7-1: Find All Functions

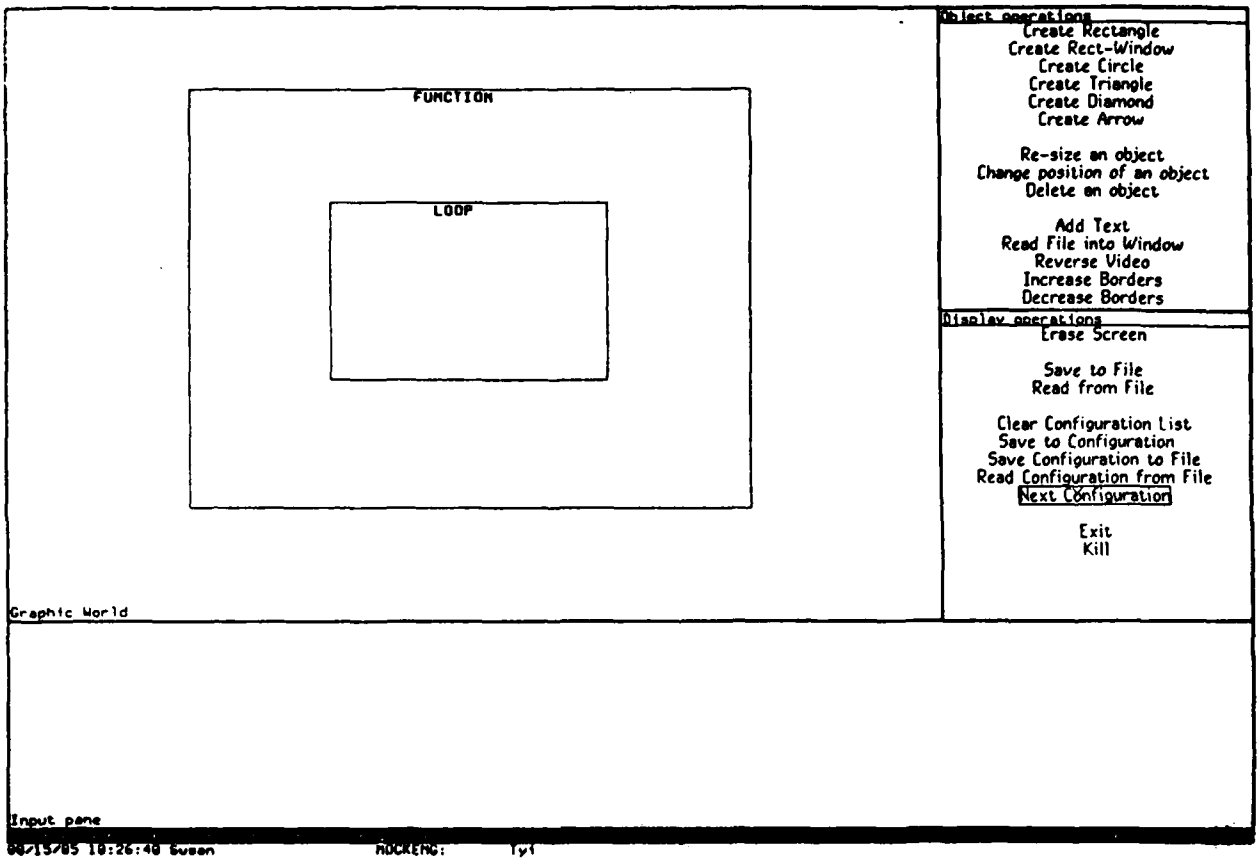


Figure 7-2: Find All Functions Containing Loops

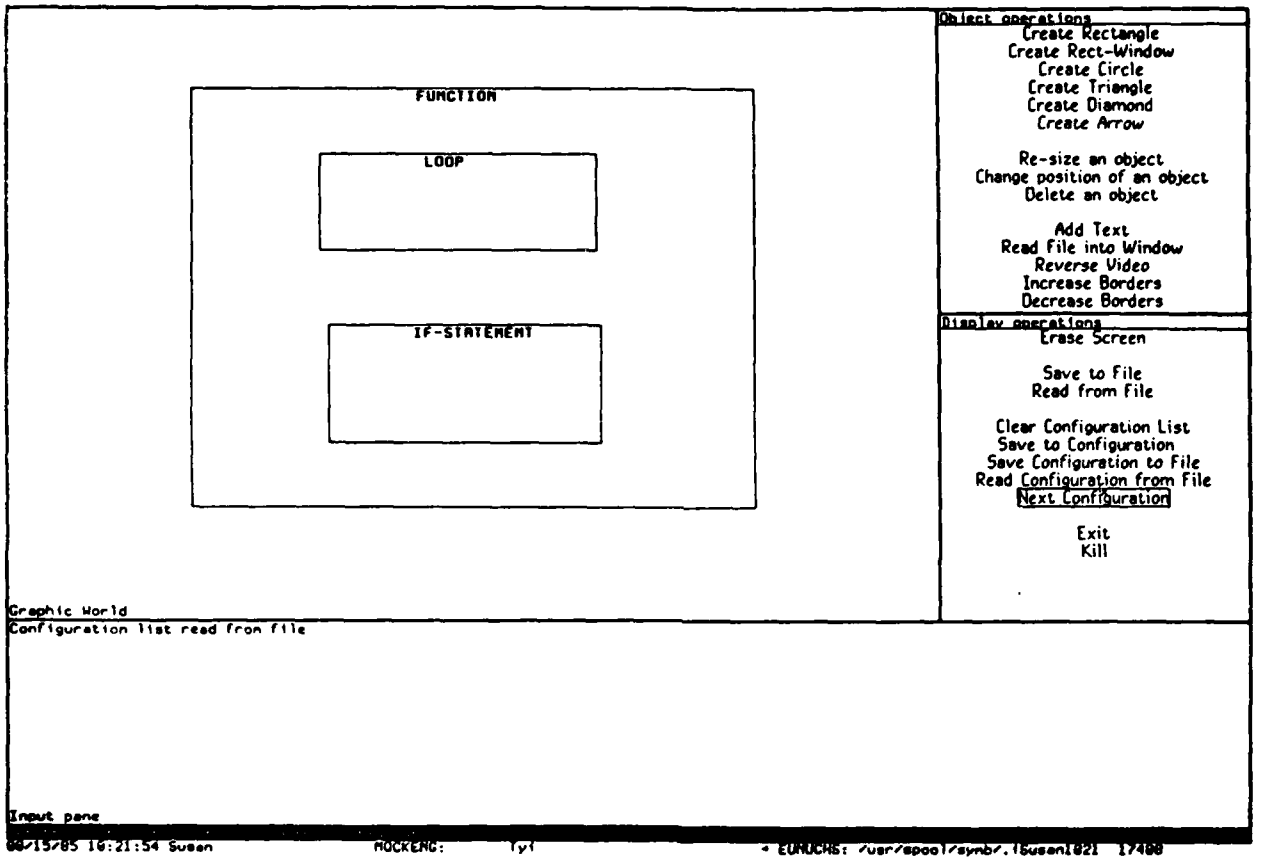


Figure 7-3: Find All Functions Containing Loops and If-Statements

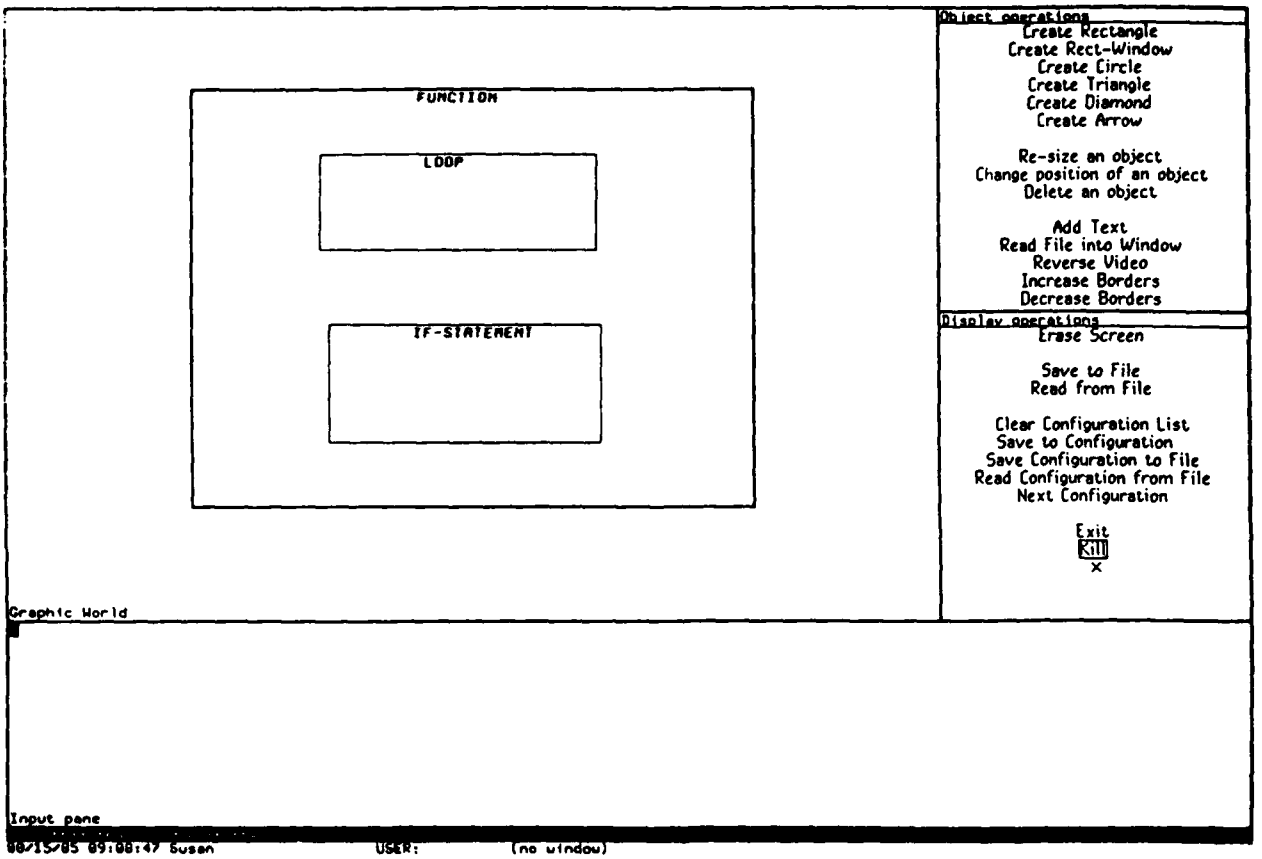


Figure 7-4: Return All Functions Matching the Query

7.3 Notes on the Design of the Display for the Picture Language

Our design considerations and the following display draw from our experience with Mock-Engine as a prototyping tool.

7.3.1 Rough Screen Design

A rough design of the editing screen is included as Figure 7-5. This design is not to scale. It is intended to illustrate the components needed at the visual interface level.

The WORK AREA of this screen is composed of panes that display the following:

- Selection menus for
  - quantification
  - parsing language
  - logical operators
  - relations
  - object types
  - focusers, highlight, ...
  - top level control
- The current construction environment.
- The current parsing of construction.
- The returnables under the current construction.
- Pop-up windows for rarely used items.

7.3.2 Additional Representation Issues

We conclude this Chapter with a brief discussion of several issues of screen representation and functionality.

1. PROPER NAMES: The user should be able to specify a proper name, and the system furnishes the type. Note that only VARIABLES and FUNCTIONS have proper names. Display of proper names looks the same as display by clicking the name slot, i.e.:

   function
   name: foo

2. CONSISTENCY OF VARIABLE LABELS: If a label is used more than once in the same configuration, it refers to the same object. If the user wishes to specify two different functions, for example, he should say FN1 and FN2.

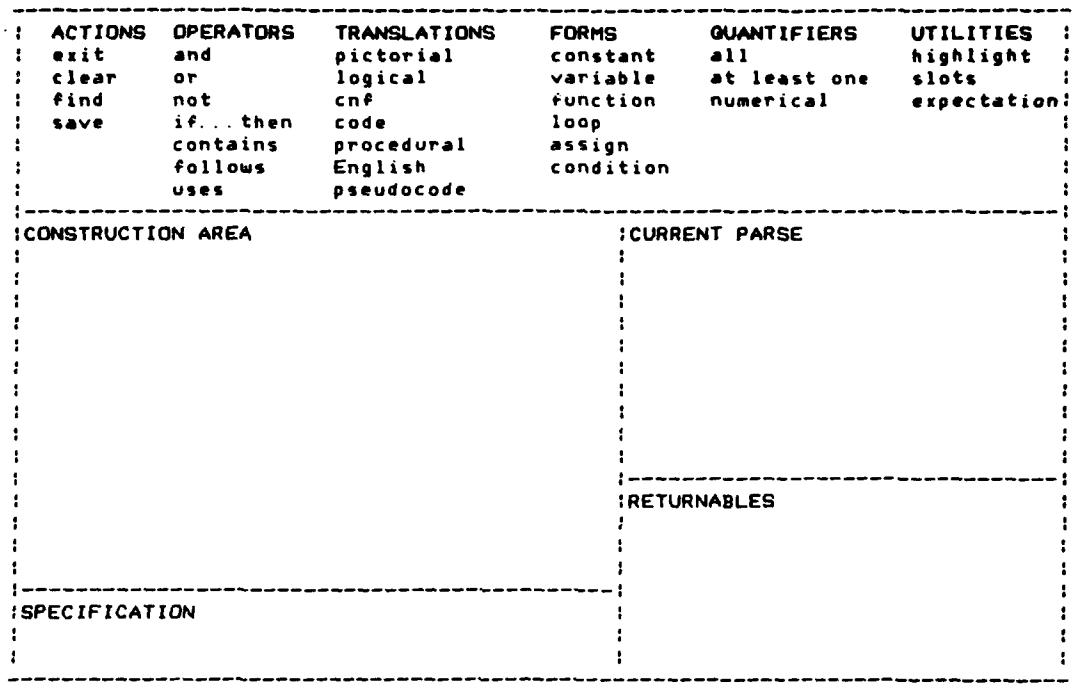


Figure 7-5: Rough Screen Design for the Visual Interface

3. OBJECT/ATTRIBUTE STRUCTURE: Each type of object has an attribute structure. It may be possible to include a clickable menu of common object types, but in any event, clicking on an object type label should pop up a menu of available attributes. Clicking on a particular attribute label should add the attribute name under the object variable, and position the cursor ready to enter a particular specification for the attribute. For example (@ is "click"):

    STEP 1.

    @ function

    STEP 2.

    function
    name: <cursor here>

    STEP 3.

    function
    name: foo <typed by user>

   The value of the attribute field called *name* is now FOO.

4. QUANTIFICATION: When an object label is entered, and the user types a CARRIAGE RETURN, a quantification field label, "#:" appears under the object label, followed by the cursor. Options are ALL, a number, THE, or COUNT. THE means the single one. COUNT means to determine how many instances are in the database by counting. Invalid specifications would give an error message, and allow another attempt.

   Non-quantified variables will default to universal quantification. The highlighted object, the focus of the search, is also assumed to be quantified universally. For example:

    Find the functions that contain a loop.

   becomes

    `(loop over all functions, find ALL loops).`

   The system should report the number of instantiations, especially if it gets large (say >20), in a warning message. The system should ask:

    There are at least 45 instances of this request. Do you want them all, or do you want to further constrain the search?

5. NEGATION: The negation of the relation CONTAINS, *DOES NOT CONTAIN* requires the existence of the container and the universal non-existence of the contained. The prototype quantification is:

```
All X Not Exists Y . (contains X Y)
```

which says that All X contain no Y. The Y does not exist. The quantifier negation can be transferred to the relation by transformation rules, resulting in

```
All X All Y . not (contains X Y)
```

which says, equivalently, that All X do not contain any Y.

This should be the default whenever the user specifies not-contains.

The representation of NOT-CONTAINS in the PL cannot cancel the containment:

```
[[ fn [lp] ]]
```

is needed, the double bracket being NOT-CONTAINS.

6. TRANSFORMATION RULES: EPM transformation rules specify alternative display forms. The primary rules are:

   TRANSPOSITION:

   ```
   a [ [b] [c] ]  <==>  [ [a b] [a c] ]
   ```

   FLEX:

   ```
   [ [a [b]] [[a] c] ] <==>  [a b] [[a] [c]]
   ```

   As well, some "automatic" transformations convert complex expressions into equivalent simple ones:

   REFERENCE: `a a <==> a`

   INVOLTION: `[[ a ]] <==> a`

   PERVASION: `[ a ] a <==> [ ] a`

   DOMINION: `a [ ] <==> [ ]`

7. MIXED VOCABULARY: These languages are available to express the user's needs:

   - Common English: Find the functions that contain loops.
   - Formal Logic: All FN Exists LP . (contains FN LP)
   - EPM: [FNx [LPx [fn [lp]]]]
   - Pseudo-code: (find FN (contains FN LP))

   Users should be able to mix these languages to a moderate degree.

- Mixed: if [fn [lp]] then fn

The display work area can be sensitive to objects that are labels, regardless of language, and try to parse them into a comprehensible whole.

8. CLICKABILITY: generally all construction should be reachable from menu selections. The exception would be proper names.

   All clicks should be SMART. When a user clicks a selection, the system should construct as much of the desired form as possible. A template would be displayed with as much information as is supported by the intelligent processor filled in. The user would complete the form. An example sequence might be:

   a. user clicks OR,

   b. [[ ] [ ]] is constructed in work area (recall the PL parsing regime),

   c. cursor goes to *: in [[*] [ ]] and waits for label entry,

   d. after CR, cursor prompts for additionals such as quantification,

   e. slots, etc., after final CR, cursor goes to next item, eg: [[foo] [*]].

9. EXPECTATIONS: The system should display its expectations whenever a request is entered. This is a form of verification.

```
Request:        ''Find the functions that contain loops.''

System:         ''The expected form of the results is:

                        (  (FN1 (LP1 LP2))
                           (FN2 (LP3))
                           ... )

                ''OK?''
```

A. Example Queries in the Picture Language

This Appendix contains the queries delineated in the AI&DS PRL Final Report TM-1014-4, entitled Design of a Pictorial Program Reference Language. Each query is expressed in three different representation languages: Predicate Calculus, LISP pseudo-code, and the formal Picture Language.

Generally, examples will be stated in four steps. First the query is expressed in Predicate Calculus. That formal specification is converted into LISP pseudo-code. The pseudo-code is converted into PL and simplified. Finally an abstract example of the results is presented.

A.1 TOP LEVEL SCHEMATIC FOR THE EXAMPLES

1. LOGICAL SENTENCE:

   The query expressed in Relational Calculus.

   Extended logic refers to the Annotated Predicate Calculus of R. Michalski.

   Note that the top level of description, (ie: All OBJ such that ...), is the universal access to the database.

2. COMPUTABLE PROGRAM:

   The LISP pseudo-code representation of the logical sentence.

   The implementation is in terms of recognizers and collectors. An encapsulating

```
(loop over all objects ...
      return list of successes)
```

   is assumed for all code.

   The function (RECOG x) looks like:

```
(= (type OBJ) 'X)
```

   Search functions will return a list of objects meeting the specification and their contents or containers. The additional information can be removed later if necessary, at the interface level.

3. PL STRUCTURE:

The derivation of the Picture Language representation from the LISP pseudo-code.

A name suffixed by an "x" is a quantifier label.

(type x 'y) is expressed merely as y.

The quantified version is formed by skolemizing and eliminating quantifiers.

Skolem functions are expressed as Sobj.

Finally, a mnumonic for Sobj is reintroduced, eg: lpe. The "e" is for existential.

The PL form is a pattern-matching template for LISP programs. For example, in Query #1:

```
[ fn $a [ lp $b ] $c ]
```

matches LISP

```
( fn any1 ( lp any2 ) any3 )
```

The $ variables match any structures including nil. The abstract "fn" matches any *true* return from (recog FN); "lp" matches any (recog LP), such as "FOR" "WHILE" etc. Finally, brackets match parentheses.

4. RESULT

An abstract representation of what the output should look like.

A.2 THE QUERY EXAMPLES

1. FIND THE FUNCTIONS THAT CONTAIN LOOPS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X . (type X 'LP) and (contains OBJ X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (collect LP into LPs)) ))
    (collect-and-return (list FN LPs)) )
```

Note that all LPs are collected in order to keep the symmetry of CONTAINS. Alternatively, non-symmetrical code would be:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (return t)) ))
    (collect-and-return FNs) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'lp) (contains obj x) ]]

==>  [ FNx [ LPx (contains fn lp) ]]
==>  (contains fn Sfn)
==>  [fn [lpe]]
```

DISCUSSION:

See NOTE A.

RESULT:

```
( (FN1 (LP1 LP2))
  (FN2 (LP3 LP4 LP5))
  (FN3 (LP6))
          ... )
```

2. FIND THE LOOPS CONTAINED IN FUNCTIONS.

LOGIC:

```
All OBJ . (type OBJ 'LP) and
  Exists X . (type X 'FN) and (contains X OBJ)
```

PROGRAM:

Same as #1

The directionality of containment is irrelevant at this level.
The interface level can take care of this distinction.

PL:

```
[ OBJx [ (type obj 'lp) Xx (type x 'fn) (contains x obj) ]]

==>  [ LPx [ FNx (contains fn lp) ]]
==>  (contains Slp lp)
==>  [fne [lp]]
```

RESULT:

```
(  (LP1 FN1)
   (LP2 FN1)
   (LP3 FN2)
   ... )
```

3. FIND ALL FUNCTIONS CONTAINING LOOPS AND IF-STATEMENTS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ X)
               and (contains OBJ Y)
```

EXTENDED LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ (X and Y))
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            do
            (if (recog LP)
                (collect into LPs))
            (if (recog IF)
                (collect into IFs)))
         (not-empty LPs)
         (not-empty IFs) )
    (collect-and-return (list FN LPs IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         (contains obj x) (contains obj y) ]]

==>  [ FNx [ LPx IFx (contains fn lp) (contains fn if) ]]
==>  (contains fn S1fn) (contains fn S2fn)
==>  [fn [lpe]] [fn [ife]]
```

Distribution converts this to the extended logic form:

```
==>  [fn [lpe ife]]
```

RESULT:

```
(  (FN1 (LP1 LP2) (IF1 IF2 IF3))
   (FN2 (LP3) (IF4 IF5))
   ... )
```

4. FIND ALL FUNCTIONS CONTAINING A LOOP FOLLOWED BY AN IF-STATEMENT.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ X)
               and (contains OBJ Y)
               and (follows Y X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over remaining-contents-of-FN
                        (if (recog IF) (save IF)) ))
                (collect into LPs-with-IF))))
    (collect-and-return (list FN LPs-with-IF)))
```

Note: the "followed by" is assumed to be implicit in the sequence of the contents-of-FN. That is, if X contains Y and Z follows Y, then X contains Z.

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         (contains obj x) (contains obj y) (follows y x) ]]

==>  [ FNx [ LPx IFx (contains fn lp) (contains fn if)
                     (follows if lp) ]]
==>  (contains fn S1fn) (contains fn S2fn)
                        (follows S1fn S2fn)
==>  [fn [lpe]] [fn [ife]]
           \_________/
```

which can be rewritten:

```
[fn [lpe-->ife]]
```

RESULT:

```
(  (FN1 (LP1 IF1))
   (FN2 (LP2 IF2) (LP3 IF3))
   ... )
```

5. FIND ALL FUNCTIONS WHICH CONTAIN LOOPS THAT CONTAIN IF-STATEMENTS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
               and (contains OBJ X)
               and (contains X Y)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over contents-of-LP
                        (if (recog IF)
                            (collect into IFs))))
                (collect into LPs-with-contained-IFs))))
    (collect-and-return (list FN LPs-with-contained-IFs)))
```

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         (contains obj x) (contains x y) ]]

==>  [ FNx [ LPx IFx (contains fn lp) (contains lp if) ]]
==>  (contains fn S1fn) (contains S1fn S2fn)
==>  [fn [lpe]] [lpe [ife]]
```

which could be rewritten:

```
[ [[fn] [lpe]] [lpe ife] ]
```

RESULT:

```
(  (FN1 (LP1 (IF1 IF2)) (LP2 (IF3)))
   (FN2 (LP3 (IF4)))
   ... )
```

6. FIND THE FUNCTION NAMED BAR.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
          (slot-value (slot-type OBJ 'name) 'BAR)
```

PROGRAM:

```
(if (and (recog FN)
         (= (slot-value FN 'name) 'BAR))
            (return (representation FN)))
```

PL:

```
[ OBJx [ (type obj 'fn)
         (slot-value (slot-type obj 'name) 'BAR) ]]

==>  (slot-value (slot-type fn 'name) 'BAR)
```

REPRESENTATION:

function
name: BAR

The above configuration is seen as a single descriptor.
Since the NAME is a fundamental reference, an abbrevation
might be appropriate, for example, fn:bar

RESULT:

Some representation of BAR.

The representation may be the code typographical representation, the frame with slots filled in, or the location of BAR relative to a context of inquiry (callers, used variables, etc.)

7. FIND ALL FUNCTIONS THAT USE THE VARIABLE FOO.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'VAR) and
            (slot-value (slot-type X 'name) 'FOO)
            and (contains OBJ X) and (uses OBJ X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop of contents-of-FN
                   (if (and (recog VAR)
                            (= (slot-value VAR name) 'FOO)
                            (uses FN VAR) )
                       (collect VAR) ))
    (collect-and-return (list FN VAR)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'var)
         (slot-value (slot-type x 'name) 'foo)
         (contains obj x) ]]

==>  [ FNx [ VARx (slot-value (slot-type var 'name) 'foo)
                               (contains fn var) ]]
==>  (slot-value (slot-type Sfn 'name) 'foo)
        (contains fn Sfn)
==>  [fn [var:foo]]
```

REPRESENTATION:

variable
name: FOO

or just var:foo

RESULT:

```
( (FN1 FOO)
  (FN2 FOO)
  (FN3 FOO)
  ... )
```

NOTES:

USE implies CONTAINS.
The variable FOO = variable named FOO.
Different VARs can have the same name.
The concept USES needs a binding-context, or focus, to be useful.
A request for a variable (which changes) named X must be syntactic.

8. FIND ALL FUNCTIONS CONTAINING LOOPS OR IF-STATEMENTS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X Y . (type X 'LP) and (type Y 'IF)
                and { (contains OBJ X) or (contains OBJ Y) }
```

EXTENDED LOGIC:

```
... (contains OBJ (X or Y))
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (or (recog LP)
                    (recog IF))
                (collect into LPs-IFs))))
   (collect-and-return (list FN LPs-IFs)))
```

PL:

```
[ OBJx [ (type obj 'fn) Xx Yx (type x 'lp) (type y 'if)
         [ [(contains obj x)] [(contains obj y)] ]  ]]

==>  [ FNx [LPx IFx [[(contains fn lp)]
                     [(contains fn if)]] ]]
==>  [[ [[(contains fn S1fn)] [(contains fn S2fn)]] ]]
==>  [ [   [fn [lpe]]    ] [   [fn [ife]]    ] ]
==>  [ fn [lpe] [ife] ]
```

RESULT:

```
(  (FN1 (LP1 LP2 IF1 LP3))
   (FN2 (IF2))
   (FN3 (IF3 LP4))
   ... )
```

9. FIND ALL FUNCTIONS CONTAINING LOOPS AND IF-STATEMENTS.

Same as #3.

Note: The explicit AND box is unnecessary.

10. FIND ALL FUNCTIONS CONTAINING LOOPS OR IF-STATEMENTS.

Same as #8.

The explicit OR box is unnecessary.

11. FIND THE FUNCTIONS THAT DO NOT CONTAIN LOOPS.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  not Exists X . (type X 'LP) and (contain OBJ X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (return nil))
            (finally return t) ))
    (collect-and-return FNs) )
```

PL:

```
[ OBJx [ (type obj 'fn)
         [ Xx (type x 'lp) (contains obj x) ]  ]]

==>  [ FNx [ [ LPx (contains fn lp) ] ]]
==>  [ FNx LPx (contains fn lp) ]
==>  [ (contains fn lp) ]
==>  [ [fn [lp]] ]
```

DISCUSSION:

See NOTE B.

RESULT:

```
(FN1 FN2 FN3 ...)
```

12. FIND THE IF-STATEMENTS NOT CONTAINED IN LOOPS.

LOGIC:

```
All OBJ . (type OBJ 'IF) and
  Not Exist X . (type X 'LP) and (contains X OBJ)
```

PROGRAM:

Assume each IF has only one location of its representation;
the program requires two passes through OBJs.

Pass 1:

```
(if (recog IF)
    (collect into IFs))
```

Pass 2:

```
(if (recog LP)
    (loop over contents-of-LP
       (if (recog IF)
           (remove IF from IFs) ))
(finally return remaining-IFs)
```

PL:

```
[ OBJx [ (type obj 'if) [ Xx  (type x 'lp)
                              (contains x obj) ] ]]

==>  [ IFx [ [ LPx  (contains lp if) ] ]]
==>  [ IFx LPx (contains lp if) ]
==>  [ [lp [if]] ]
```

RESULT:

```
(IF1 IF2 IF3 ...)
```

DISCUSSION:

See NOTE C.

13. FIND THE FUNCTIONS WHICH HAVE A LOOP NOT FOLLOWED BY AN IF-STATEMENT.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exists X . (type X 'LP) and (contains OBJ X) and
    if Exists Y . (type Y 'IF) and (contains OBJ Y)
    then not (follows Y X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog LP)
                (if (loop over remaining-contents-of-FN
                          (if (recog IF)
                              (return nil))
                          (finally return t) )
                    (collect into LPs) ))))
    (collect-and-return (list FN LPs)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'lp) (contains obj x)
         [ Yx (type y 'if) (contains obj y)
           [ [ (follows y x) ]  ]] ]]

==>  [ FNx [ LPx (contains fn lp)
               [ IFx (contains fn if)
                     [[ (follows if lp) ]] ] ] ]
==>  [[ (contains fn Sfn)
        [(contains fn if)(follows if Sfn)] ]]
==>  [fn [lpe]] [ [fn [if]] lpe-->if ]
```

RESULT:

```
(  (FN1 LP1 LP2)
   (FN2 LP3)
   ... )
```

14. FIND THE FUNCTIONS IN WHICH ALL LOOPS CONTAIN IF-STATEMENTS.

That is, find the functions that contain loops such that every contained loop contains an if-statement. See NOTE D.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  All X . (type X 'LP) and (contains OBJ X) and
    Exist Y . (type Y 'IF) and (contains X Y)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over contents-of-LP
                        (if (recog IF)
                            (collect into IFs)) ))
                (collect into LPs+IFs)) ))
    (collect-and-return (list FN LPs+IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn)
         [ Xx [ (type x 'lp) (contains obj x)
                Yx (type y 'if) (contains x y) ]] ]]

==>  [ FNx [[ LPx
              [ (contains fn lp) IFx (contains lp if) ]] ]]
==>  [ [ (contains fn lp) (contains lp Sfnlp) ]]
==>  [fn [lp]] [lp [ife]]
```

RESULT:

```
(  (FN1 (LP1 IF1) (LP2 IF2 IF3))
   (FN2 (LP3 IF4 IF5))
   ... )
```

15. FIND THE FUNCTIONS WHICH CONTAIN 2 LOOPS THAT CONTAIN IF-STATEMENTS.

Note: cardinality can be treated the same as any quantifier.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'LP) and (cardinality X 2)
                         and (contains OBJ X) and
    Exist Y . (type Y 'IF) and (contains X Y)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog LP)
                     (loop over contents-of-LP
                        (if (recog IF)
                            (collect into IFs)) ))
                (collect into LPs+IFs)) )
         (= (number-of LPs+IFs) 2) )
    (collect-and-return (list FN LPs+IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn)
         Xx (type x 'lp) (cardinality x 2) (contains obj x)
         Yx (type y 'if) (contains x y) ]]
==>  [ FNx [ LPx(2) (contains fn lp)
             IFx (contains lp if) ]]
==>  (contains fn lpe2) (contains lpe2 ife)
==>  [fn [lpe2]] [lpe2 [ife]]
```

RESULT:

```
( (FN1 (LP1 IF1) (LP2 IF2 IF3))
  (FN2 (LP3 IF4) (LP4 IF5))
  ... )
```

16. FIND THE FUNCTIONS WHICH CONTAIN AN IF-STATEMENT NOT CONTAINED IN A LOOP.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'IF) and (contains OBJ X) and
    not Exist Y . (type Y 'LP) and (contains OBJ Y)
                                   and (contains Y X)
```

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (recog IF)
                (collect into IFs)) ))
    (collect-and-return (list FN IFs)) )
```

PL:

```
[ OBJx [ (type obj 'fn) Xx (type x 'if) (contains obj x)
         [ Yx (type y 'lp) (contains obj y)
                           (contains y x) ] ]]

==>  [ FNx [ IFx (contains fn if)
               [ LPx (contains fn lp) (contains lp if) ] ]]
==>  (contains fn Sfn)
       [ (contains fn lp) (contains lp Sfn) ]
==>  [fn [ife]] [ [fn [lp]] [lp [ife]] ]
```

NOTE:

For #16 to be a meaningful query, the semantics of CONTAINS must include the idea of deep and shallow containment.

The PL example is a statement of the intransitivity of containment. Abstractly:

[a [b]] & [b [c]] imply [[a [c]]]

RESULT:

```
(  (FN1 (IF1 IF2))
   (FN2 (IF3))
   ... )
```

17. FIND THOSE FUNCTIONS THAT USE SOME VARIABLE BEFORE SETTING THAT VARIABLE.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'VAR) and (uses OBJ X)
                          and (sets OBJ X)
                          and (follows set use)
```

DISCUSSION:

See query #18.

18. FIND THOSE FUNCTIONS IN WHICH A VARIABLE IS NOT SET BEFORE THAT VARIABLE IS USED.

LOGIC:

```
All OBJ . (type OBJ 'FN) and
  Exist X . (type X 'VAR) and (uses OBJ X) and
    if (sets OBJ X) then (follows set use)
```

DISCUSSION:

The FOLLOWS construct has events rather than objects as arguments. Assuming that the events can be objectified, and that FOLLOWS can be dynamic, these translations occur:

PROGRAM:

```
(if (and (recog FN)
         (loop over contents-of-FN
            (if (and (recog VAR)
                     (uses FN VAR)
                     (if (and (sets FN VAR)
                              (follows set use) )
                         (collect VAR into VARs) ))) ))
    (collect-and-return (list FN VARs)) )
```

This assumes that the contents contain all used things. The semantics of CONTAINS will have to distinguish between shallow and deep containment.

PL:

```
[ OBJx (type obj 'fn)
        [ Xx (type x 'var) (uses obj x)
             [ (sets obj x) [ (follows set use) ]] ]]

==>  [ FNx [ VARx (uses fn var)
                  [ (sets fn var) [ (follows set use) ]] ]]
==>  (uses fn Sfn) [ (sets fn Sfn) [ (follows set use) ]]
==>  (uses fn vare) [ (sets fn vare) [ use-->set ]]
```

NOTES:

FOLLOWS is a functional.

Assignment (sets) is itself a type of function, so the only FN that both USES and SETS is assignment.

Assignment has a degenerate use of FOLLOWS.

If FN is a procedure, the meanings of USE and SET need clarification, like:

```
All OBJ . (type OBJ 'procedure) and
  Exist X Y . (type X 'var) and (type Y 'fn) and
              (contains OBJ Y) and (contains Y X) and
              (uses Y X) and
    if Exist Z . (type Z 'assign) and (contains Z X)
                                  and (sets Z X)
       then (follows Z Y)
```

Rewritten:

```
All PROC Exist VAR FN .
  (contains PROC FN) and (contains FN VAR)
                     and (uses FN VAR)
  and if Exist ASSIGN . (contains ASSIGN VAR)
                        and (sets ASSIGN VAR)
      then (follows ASSIGN FN)
```

NOTES ON QUERIES:

1. The construction of a logical specification (and code) from the request takes these steps (reversing the development in the example):

   Request:

   ```
   [fn [lpe]]
   ```

   To relations:

   ```
   (contains fn lpe)
   ```

   To type identifiers:

   ```
   (type obj 'fn) (type Sobj 'lp) (contains obj Sobj)
   ```

   To conjunct/disjunct:

   ```
   (type obj 'fn) & (type Sobj 'lp) & (contains obj Sobj)
   ```

   To elemental code:

   ```
   (if (and (type obj 'fn)
            (type Sobj 'lp)
            (contains obj Sobj) )
       (list (obj Sobj)) )
   ```

   To quantified code:

   ```
   (LOOP OVER ALL OBJ
      (LOOP OVER ALL SOBJ IN OBJ
           (if (and (type obj 'fn)
                    (type Sobj 'lp)
                    (contains obj Sobj) )
               (COLLECT SOBJs) )
           (COLLECT-AND-RETURN
               (list (obj SOBJs)) ) )
   ```

2. Since CONTAINS is an elementary relation, it is confusing to simplify away the containment, by using the rule of double negation, [[a]] ==> a.

   ```
   [[ fn [lp] ]]  =/=>  fn [lp]
   ```

   The left-hand-side reads FN AND NOT LP, which is not what we want to express.

   The representation of NOT CONTAINS be an elaboration of CONTAINS. This could be the double bracket, or it could be a highlight of the entire containment structure.

3. The PL form here is the same as in #11, but the item returned is different. Non-existent objects cannot be returned, so no confusion can result. However the information about what to return must still be displayed. In general, the object that organizes the returned values (the first one in the result list) should be highlighted in the display.

   In the abstract form:

   ```
   [ a [ b [c] ] ]
   ```

   one item will be special. Let the highlight be ASTERISKS here:

   ```
   [ a [ *b* [c] ] ]
   ```

   which means that All B are to be returned. Requests are Universal rather than Existential, unless explicitly labeled otherwise. (The request "Find a function with a loop" isn't highly motivated.)

   So the default quantification of the asterisked token is ALL.

   Although CONTAINS can be treated symmetrically for its two arguments, NOT-CONTAINS is not symmetric. Specifically,

   A contains B
   B contained-in A
   A not-contains B

   are all determined by looking inside of A.

   B not-contained-in A

   is determined by looking both inside (to assure B is not there) and outside (to find the B) of A.

4. ELLIPSIS and AMBIGUITY: Natural English is often not sufficiently specific to parse into a formal language. The Instructability strategy should be used in these cases.

   Psychologically, all requested objects should assume existence. #14 could be interpreted "if the FN contains a LP then the LP contains an IF." Under this interpretation, FNs that do not contain LPs would be returned, as well as FNs with LPs with IFs. The former group should be requested separately (its hard to imagine wanting both groups for the same reason).

B. Use of the Mock Engine

B.1 Starting the System

To begin using the Mock-Engine (henceforth known as ME), the user should type a sequence of commands to the Symbolics LISP Machine:

- (load "A:>susan>mockeng>mocksys.lisp"): This command will load the defsystem for the ME.
- (make-system 'mockeng :noconfirm): This command will load all of the binary files that compose the ME. Ignore the notice that there is an unknown special instance variable, zwei:*last-file-name-typed*.
- (pkg-goto 'mockeng): Change the package to mockeng.
- (make-mock): This command will turn the mouse from its usual arrow form to a top-left corner of a rectangle form. Position the mouse blinker in the top left-hand corner of the screen and click left. Next, move the mouse until the elastic window fills the entire screen and click left again. This will be the window used for the ME interactions. Control will be returned to the user in the lisp listener.
- (top-level): The ME frame will be exposed and the user can begin making interesting drawings.

B.2 General Screen Layout

The ME screen is divided into four windows, as can be seen in Figure B-1. The large window found in the top-left position is the graphic display window that will contain the graphic picture being developed. The window underneath that is the interaction window. The user will be prompted in this window when either an action or some input is needed. There are two menu windows. The one on the top right of the screen is the object menu. It contains the commands that operate on the individual objects, both in their creation and later manipulation. The bottom menu contains commands that operate on the screen as a whole, including clearing the screen or saving its contents.

Following is a more complete definition of the menu choices and their consequences:

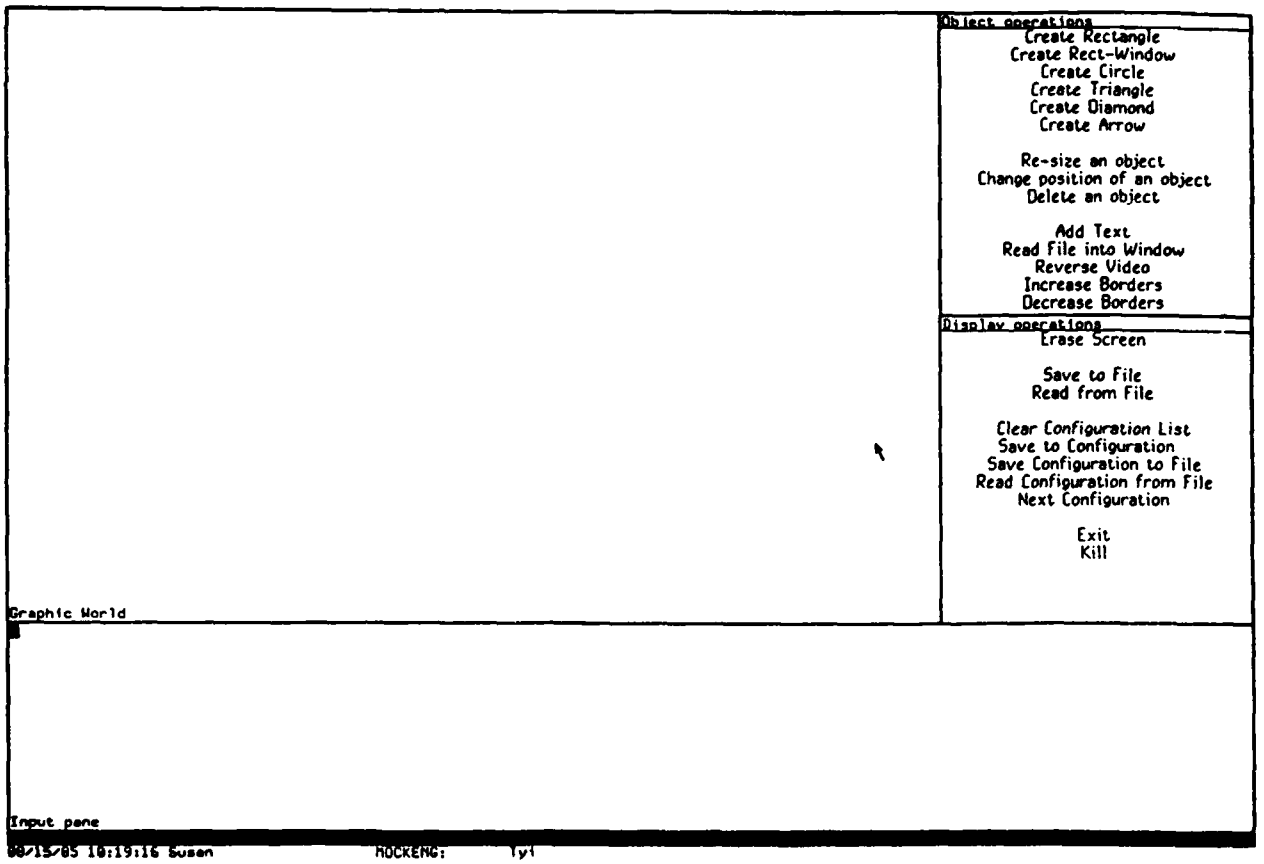


Figure B-1: Screen Design

B.3 Object Menu

Figure B-2 is an example of the Mock Engine in use. The object menu consists of the following commands and usages. (You will see a short explanation of the action of the menu choice in the mouse documentation line.) In any of the create-object options, a small object will be created that the user will drag around on the screen and initially place by a left-click. The first list of commands contains those that create objects.

- Create Rectangle: creates a new rectangle (note: this is just a basic rectangle, not an editable one)

- Create Rect-Window: creates a "rectangle-window". A left-click on this item creates a window whose superior is the entire graphics window. A right-click is needed to create a window inside an already existing rectangular-window. In this case, the user will first need to select the superior window before setting the position of the new window.

- Create Circle: creates a circle

- Create Triangle: creates a triangle

- Create Diamond: creates a diamond

- Create Arrow: creates an arrow. The user will first be prompted for the head object to use for the arrow and then for the tail object for the arrow. The arrow will be drawn from the midpoint of an edge of the head object to the midpoint of an edge of the tail object.

The next set of commands are those that operate on any of the object types. The selection of an object is accomplished by moving the mouse blinker over the desired object. The system will indicate that the mouse blinker is over an existing object by drawing a rectangular outline around it.

- RE-SIZE AN OBJECT: changes the size of an existing object. The user will first be asked to select the object to be modified and will then be given a small object-like thing to expand or contract until the desired size is obtained.

- CHANGE POSITION OF AN OBJECT: move an object to a new position. The user will select the object to be moved and then be able to drag it around to its new position.

- DELETE AN OBJECT: delete an object. The user will select the object that is about to die and then it will cease to exist.

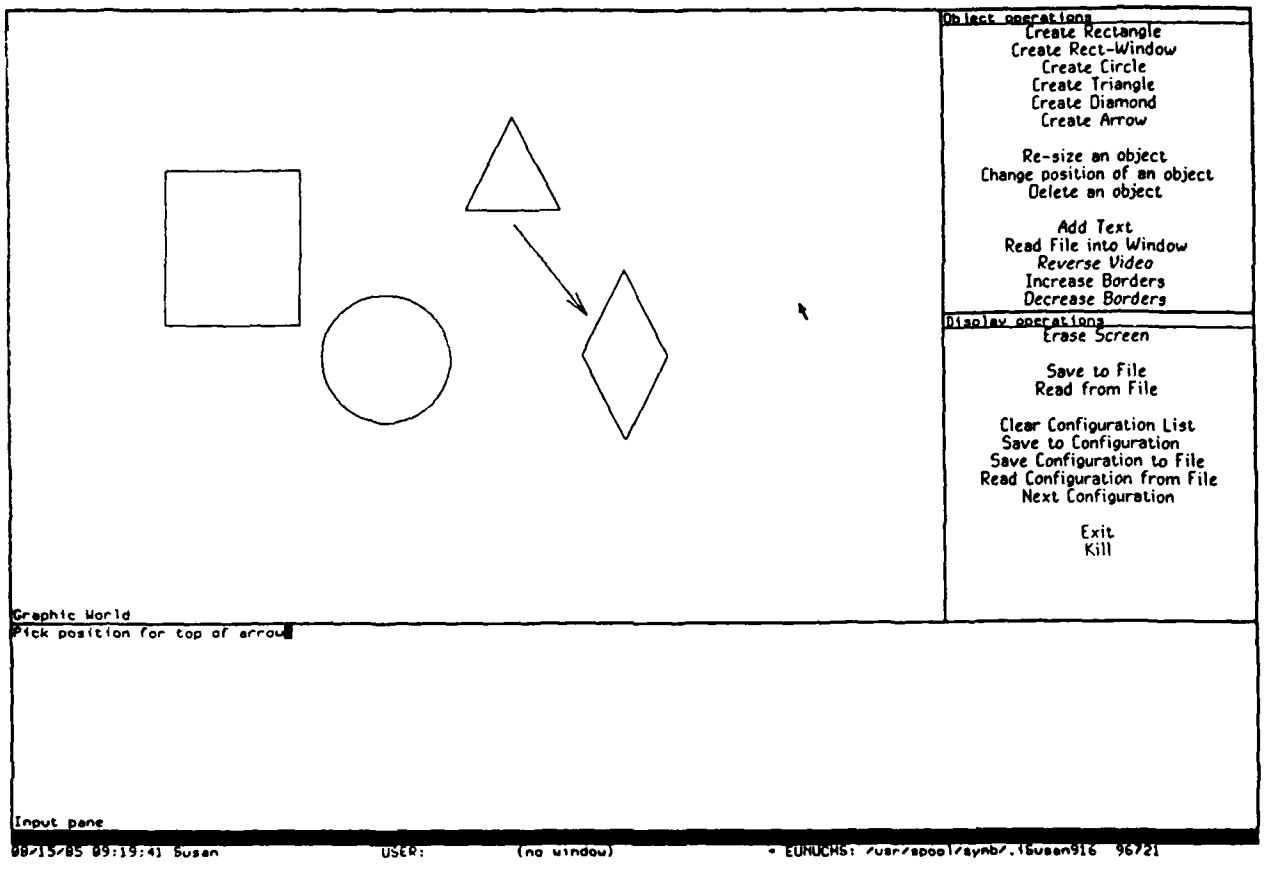


Figure B-2: Sample Mock Engine Useage

The commands that follow are those that operate only on the "rectangular-windows".

- ADD TEXT: add text to a window. The user will select a window and then be able to type text into that window using all of the favorite EMACS commands. Once satisfied with the window's contents, the user should hit the <END> key.

- READ FILE INTO WINDOW: read a file into a window. The user will first be prompted to select a window and then will be asked to input the name of a file to be read into that window.

- REVERSE VIDEO: reverse the video on a window. The user will select a window whose background color should reverse itself from white to gray or from gray to white.

- INCREASE BORDERS: increase the border width of a window. The user will select a window whose border width should be doubled.

- DECREASE BORDERS: decrease the border width of a window. The user will select a window whose border will return to the default width.

B.4 Screen Menu

The screen menu consists of those commands that are specific to operation on the entire screen. They are:

- ERASE SCREEN: clear the graphics window. This command both clears the graphics window and deletes all of the objects that it may have contained.

- SAVE TO FILE: save the current graphic display on a file. The user will be prompted for a file name on which to save the graphic objects that form the current picture.

- READ FROM FILE: read the graphic display from a file. The user will be prompted for a file name from which to read one graphic display.

The next group of commands operate on what is called the "configuration list." This list is an internal list that can contain different graphic configurations for saving and later display. The user will use this list to save a group of graphic displays. He will then save this grouping on a file and later be able to read that file back into the "configuration list" for a slide-presentation effect of flipping through a series of pre-defined displays.

- CLEAR CONFIGURATION LIST: clear the configuration list. Clear the configuration list of its contents.

- SAVE TO CONFIGURATION: save the current graphic display in the configuration list. The objects in the current screen display will be saved.

- SAVE CONFIGURATION TO FILE: save the current configuration list to a file. The user will be prompted for a file name on which to save a group of screen displays.

- READ CONFIGURATION FROM FILE: read the configuration list from a file. The user will supply a file name of pre-designed displays. The file will then be read into the configuration list.

- NEXT CONFIGURATION: get the next configuration the configuration list. The next display that was stored on the configuration list will be displayed on the graphics window.

The last two commands are strictly general system commands.

- EXIT: exit the program. Returns control to the lisp system. The user can return to the lisp listener by typing a <select> <L>. Later, if he wants to return to the ME, he will type (top-level) and will be placed back into the state that he left.

- KILL: kill the window. Kill the ME and exit the program.

END

FILMED

3-86

DTIC